UNIVERSIDAD
COMPLUTENSE
MADRID

DESIGN OF A TEST PLATFORM FOR STUDYING RADIATION EFFECTS ON SPI FRAM AND I²C CMOS NON-VOLATILE MEMORIES

RAÚL GIL FERNÁNDEZ – DANIEL LEÓN GONZÁLEZ

TRABAJO DE FIN DE GRADO – GRADO EN INGENERÍA INFORMÁTICA 2018/19
FACULTAD DE INFORMÁTICA – UNIVERSIDAD COMPLUTENSE DE MADRID
JUNE 2019 ORDINARY CALL

DIRECTED BY:
JUAN ANTONIO CLEMENTE BARREIRA
JUAN CARLOS FABERO JIMÉNEZ

Index

SUMMARY

Electronic components are often exposed to external radiation that may interfere with their normal operation. This becomes particularly relevant for high-scale of integration parts and when said components are used in harsh environments such as in satellites and space ships, aviation, military applications or in heavy and automotive industries. Several design and construction techniques have been developed to overcome, or at least minimize, the impact of external factors on electronic components, so ESD (Electro-Static Discharge), EMI (Electro-Magnetic Interference) or RFI (Radio-Frequency Interference) protection is usually built-in into mission-critical electronic components like microcontrollers, FPGAs or memory chips.

The Computer Architecture Department at FDI is carrying out the READAR-II project with the aim of developing techniques to improve the resistance of different types of components used in space missions. In particular, and as a first step, the study on semiconductor-based memories. This BSc. Thesis goal lies within the READAR-II research project objective and focuses on building a test platform to evaluate the radiation effects on non-volatile memories.

Keywords

Radiation effects on non-volatile memories, fault tolerance, memory test platform, READAR-II.

RESUMEN

Los componentes electrónicos suelen estar expuestos a radiaciones externas que pueden interferir con su funcionamiento normal. Esto es particularmente relevante cuando componentes con alta escala de integración se utilizan en entornos hostiles, como en satélites y naves espaciales, aviación, aplicaciones militares, industria pesada o en la industria automotriz. Se han desarrollado varias técnicas de diseño y construcción para superar, o al menos minimizar, el impacto de los factores externos en los componentes electrónicos, por lo que la protección frente a ESD (Descarga Electroestática), EMI (Interferencia Electromagnética) o RFI (Interferencia de Radiofrecuencia) suele estar incorporada en los componentes electrónicos de misión crítica como los microcontroladores, dispositivos de lógica reconfigurable (FPGA) o chips de memoria.

El departamento de Arquitectura de Computadores y Automática de la FDI está llevando a cabo el proyecto de investigación READAR-II[1] con el objetivo de desarrollar técnicas para mejorar la resistencia de diferentes tipos de componentes utilizados en misiones espaciales, en particular y como primer paso, se realiza el estudio sobre memorias

[1] Hardware and Software Strategies for Radiation-Induced Error Analysis, Detection and Recovery on Spacecraft Digital Systems II

basadas en semiconductor. Este trabajo de fin de grado se engloba dentro del objetivo del proyecto READAR-II y se centra en la creación de una plataforma de pruebas que permita evaluar los efectos de la radiación sobre memorias no volátiles.

Palabras clave

Efectos de radiación en memorias no volátiles, tolerancia a fallos, plataforma de pruebas para memorias, READAR-II.

CHAPTER 1: INTRODUCTION

This chapter describes the specific goals for this thesis, as well as the overall scope and contribution of each of the members.

BSc. Thesis – Specific goals

The goal of this project is to create a hardware/software test platform for studying radiation effects on FRAM[2] and CMOS[3] non-volatile memories. To this end, a microcontroller-based circuit is designed so it can interact with a PC user interface and control all operations performed on the memory chips. Since the platform is meant to be used in front of a particle accelerator, the microcontroller part should be at a safe distance from the irradiated memories. This could be achieved by using a daughter board for placing the memory chips.

The microcontroller board is responsible for receiving the test actions from the PC, writing to the memory and reading back as instructed and communicate the results back to the PC for easy reading. As a secondary goal, the microcontroller should also control the voltage fed to the memories to simulate different power supply scenarios. This technique is known as Dynamic Voltage Scaling (DVS) and it is usually adopted to save power when the device is idle. The preferred microcontroller platform is Arduino Due. The daughter memory board should use a RJ45 header, thus using eight wires, to connect to the main microcontroller board. It must also allow for an easy exchanging of memory chips by using ZIF[4] sockets.

The PC configuration and operation software should be platform-independent, and it must also feature a user-friendly graphical interface. Its design includes two main components, a program running over the main microcontroller board and a Graphical User Interface (GUI) based on JAVA Swing working in parallel on a PC. The main advantage is that, as the Arduino program is automatically run when the board is switched on, it is just necessary to launch the GUI to start working. This significantly reduces the time needed while changing the radiation focus over memories.

Work team and methodology

The project has been divided into two working packages, hardware-related and software-related. Daniel León González was in charge of the Printed Circuit Board (PCB) design and construction, as well as the communication between the main microcontroller board and the daughter board containing the sockets and memories. Raúl Gil Fernández developed the microcontroller firmware running in the Arduino Due platform and the Java GUI, abstracting the main testing functionality to the user.

[2] Serial Peripheral Interface - Ferromagnetic RAM
[3] Inter Integrated Circuit - Complementary metal oxide semiconductor
[4] Zero Insertion Force

The first step taken was clarifying the project goals and tasks to be done. There was an intensive communication and feedback between both team members as well as with the project directors. Bi-weekly meetings were held, customizing Scrum[5] methodology, to set the two weeks sprint goals and review the ongoing status of the project.

The design and creation of the main and daughter boards were first accomplished within two months from the beginning of the project. The developing of the software core structures was accomplished during the same timeframe. The goal was to have a ready-to-test software once the PCB manufacturer delivered the boards.

Memory part availability was first identified as a risk and it actually became a problem. It was not until two months prior to the Thesis delivery that we could have a full setup ready to integrate. Some adaptations were required on the board since the memories manufacturer (Cypress), failed to provide the advertised encapsulation.

Over a month period, Raúl and Daniel fine-tuned the low-level protocol implementations and Raúl did also develop the high-level integration software and the PC Java GUI. Structural and black-box tests were performed to ensure a smooth integration between hardware and software.

[5] Scrum methodology: http://www.scrum.org

CHAPTER 2: BASE TECHNOLOGY AND PROTOCOLS

This chapter describes the technology of the used components and how they communicate with the Arduino/PC based test platform. It covers the three used memories:

Cypress CY14B101J – nvSRAM memory
Cypress CY15B102Q and CY15B104Q – FRAM memories

and the three serial communication protocols used in the project:

I^2C – Inter-Integrated Circuit
SPI – Serial Peripheral Interface
Asynchronous serial

Serial communication is the most pervasive communication scheme used today. It presents several advantages when compared to parallel communication, such as circuit simplicity, component pin usage and higher speeds. Since serial communication is less susceptible to autoinductances than parallel, it can achieve much faster transfer rates. In fact, all communication protocols used in this work are serial communication protocols, either synchronous or asynchronous. Other relevant serial protocols are USB, SATA, JTAG, ModBus, CAN, Midi or Microwire.

For these three communication protocols, the protocol itself is described, highlighting the most relevant characteristics, and the specific implementation for the devices used in the project is detailed. When referring to the number of wires used by each communication protocol, a common reference ground is assumed to exist already between the transmitter and the receiver.

nvSRAM Memory: Cypress CY14B101J [6]

This part is a 1Mbit, 128Kx8 non-volatile static RAM that works like a normal RAM but provides a feature called “Quantum Trap”, which can be triggered by software commands, by a hardware pin and/or automatically at power-down. This “Quantum Trap” is the actual non-volatile component of the memory chip with a 1 million write cycles. By not storing every write on the non-volatile part, the endurance of this memory is greatly extended.

The most significant features and rates of this part are:

- Infinite read, write and recall operations.
- 1 million write cycles to Quantum Trap.
- 20 years data retention.
- 2.7V to 3.6V operation.

[6] CY14B101J Datasheet: https://www.cypress.com/file/44701/download

- 8µA (sleep), 150µA (standby) and 1mA (operation) current consumption.
- SOIC-8 150mil. package.
- I^2C serial interface with zero delays. Speeds from 100KHz up to 3.4MHz.

The block diagram for this part is depicted in *Figure 1*.

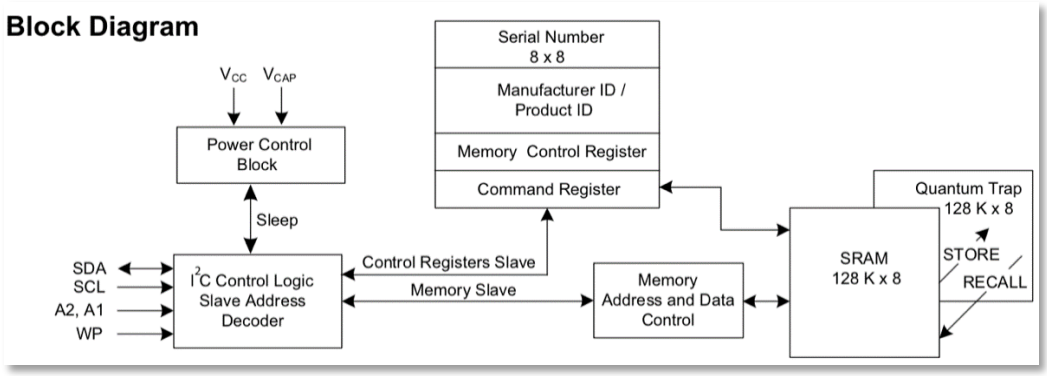

Figure 1: Block diagram for nvSRAM CY14B101J (from datasheet)

The offering of this memory family includes the C14C101J and C14E101J, covering different operating voltages. Although the datasheet mentions a SOIC16 variant of the memory, it is not commercially available at the time of writing this document.

FRAM Memories: Cypress CY15B102Q[7] and CY15B104Q[8]

These parts are based on a ferroelectric layer that provides its non-volatile feature. These two memories provide 2Mbit (128Kx8) and 4Mbit (256Kx8) respectively, and they have significant internal construction differences that prove interesting for the goal of the study; thus, the 2Mbit memory is implemented using 2T2C technology, whereas the 4Mbit one is implemented using 1T1C cells. The internal configuration for these technologies makes them have a potentially different behavior when exposed to radiation.

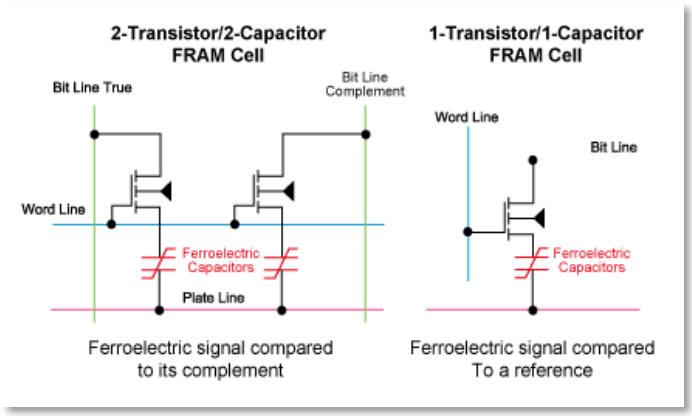

Figure 2: Schematic for 2T2C and 1T1C FRAM memories (Ramtron International)

[7] CY15B102Q Datasheet: https://www.cypress.com/file/175731/download
[8] CY15B104Q Datasheet: https://www.cypress.com/file/209146/download

The block diagram in *Figure 3* illustrates the organization of the 2Mbit memory. The diagram for the 4Mbit memory only changes the FRAM array size.

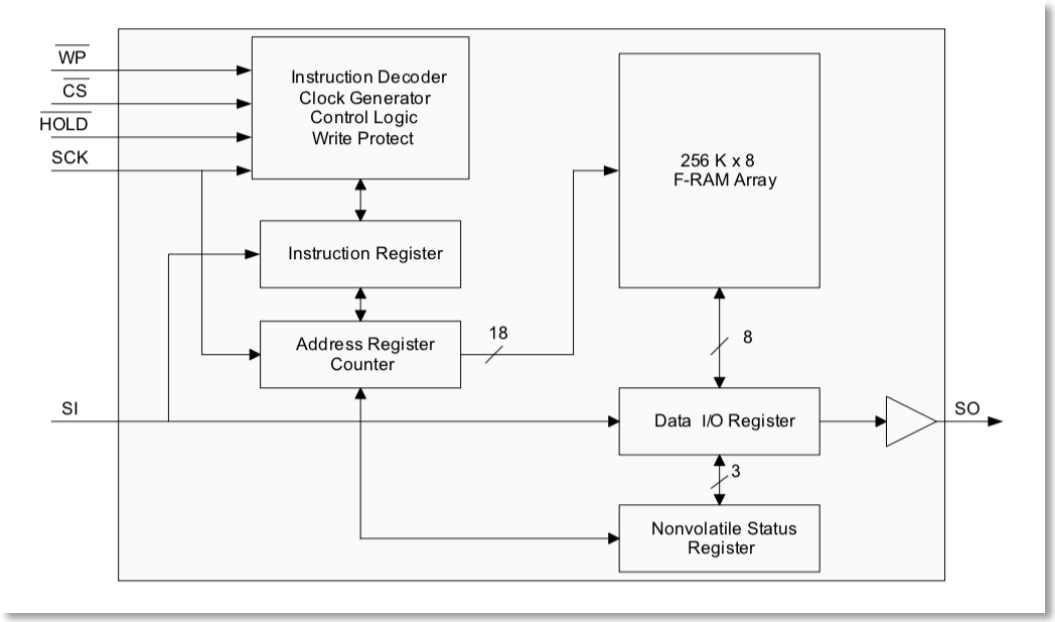

Figure 3: Block diagram for FRAM CY15B102Q (from datasheet)

Note the Data I/O register is a shift register implementing the SPI protocol described later in this document.

The most significant features and rates of these parts are:

- 10 trillion (10^{13}) read/writes (CY15B102Q) 100 trillion (10^{14}) (CY15B104Q).
- 121 years data retention (CY15B102Q), 151 years (CY15B104Q).
- 2.0V to 3.6V operation.
- 20µA (sleep), 750µA (standby) and 5mA (operation) current consumption (CY15B102Q).
- 3µA (sleep), 100µA (standby) and 300µA (operation) current consumption (CY15B104Q).
- SOIC-8 package.
- SPI serial interface up to 25 MHz (CY15B102Q) and up to 40 MHz (CY15B104Q).

I²C: Inter Integrated Circuit [9] [10]

I²C is a serial protocol invented in 1982 by Philips, targeting the interconnection of multiple physically close circuits, at moderate speeds, using only two wires, SCL (Clock) and SDA (Data).

[9] I²C bus specification and user Manual: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[10] AN10216-01 I²C Manual: https://www.nxp.com/docs/en/application-note/AN10216.pdf

It is a synchronous protocol allowing for multiple master and slave devices to be connected to the same bus, creating an n-to-m communication topology. Signals are driven by open-drain / open-collector endpoints, thus requiring bus pull-ups in both SCL and SDA lines.

Addressing

Since all devices are connected to the same bus, those acting as slaves must have a unique address to identify themselves. This address is usually composed by a 7-bit signal, followed by a Read/Write bit. I²C busses may also use a 10-bit address scheme, however, none of the devices used for this study use 10-bit addressing. Addresses are managed by the owner of the patent, currently NXP, and acquiring a unique address or a set of addresses requires a fee.

Devices can have a fully fixed or a partly fixed address. For the latter, a set of dedicated pins should be driven high or low to complete the unique address in the bus, enabling the co-existence of several devices with the same part-number in the bus.

CY14B101J memories have the following I²C addresses (X: selectable 0 or 1):

1010 XX A16 : For memory operations
0011 XX X : For register operations

The configuration for this work sets the variable address bits to '0'.

The inclusion of A16 in the address of the device is somewhat on the edge of the specification, but it responds to an optimization goal. Since the size of these memories is 128KB, they use 17 bits for address. This would have required the memory location address to be sent using 3 bytes but, by moving A16 to the "device address" byte, it only requires 2 bytes, A0-A15 to be sent for memory operations. On the other hand, this causes that just 4 CY14B101J chips may be directly attached to the same I²C bus, instead of the potential 8 chips.

Figure 4 details the addressing scheme used by CY14B101J memories.

Figure 4: CY14B101J I²C addressing, Memory and registers (from datasheet)

There are reserved addresses that require different responses in the bus. These are listed in the I²C bus specification document[11], and shown in the table below.

[11] I²C bus specification and user Manual: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

Slave Address	R/W bit	Description
0000 000	0	General call address
0000 000	1	Start byte
0000 001	X	CBUS address
0000 010	X	Reserved for different bus format
0000 011	X	Reserved for future purposes
0000 1XX	X	Hs-mode Master code
1111 1XX	1	Device ID
1111 0XX	X	10-bit addressing mode

Table 1: Reserved I²C addresses

Bus read and write operations

The I^2C bus is managed by the current master device, which drives the clock line and either writes to the bus or reads from it. Slave devices can only access the data line of the bus when previously instructed by the master and only on the clock signals driven by the master. Data communication is half duplex, since both operations (reads and writes) are transmitted over the SDA line.

In multi-master environments, there are bus arbitration procedures in place that prevent multiple devices from acting as bus-master at the same time. These are based on the monitoring of the SDA line when the master proposes a '1' if the signal is '0', which would mean that another master is driving the line low. In this case, the affected master withdraws from mastering the bus and reverts back to slave mode. Our setup does not include multi-master configuration, since it is a simple 1-Master, 1-Slave circuit.

Each message driven by a master must start with a "start condition" and end with a "stop condition". There could be more than one start condition in a single message, which is called a "repeated start" and it effectively re-starts the communication without releasing the bus, hence preventing other masters from claiming control for themselves. A "start condition" is signaled by pulling the SDA line low and then pulling the SCL line low. A "stop condition" is signaled by releasing the SCL line (to high) and then releasing the SDA line. In both cases, a transition in SDA happens when SCL is high. On the other hand, SDA for normal communications only changes when SCL is low. This behavior is clearly appreciated in *Figure 5*.

Devices acknowledge the reception of the message in write operations by driving the SDA line low right after each data (or address) communication to them. Masters acknowledge the reception of messages got back from read operations in the same way. They do not acknowledge the last message after issuing a "stop condition".

A message is thus composed of the following elements:

- Start condition (or repeated start).
- Device ID of the recipient of the message + R/W bit.
- Data bytes + ACK (as many as required by the device, this is device-dependent).

- Stop condition (or repeated start = stop + start).

The waveform in *Figure 5* shows a typical I²C communication of address + 2 bytes.

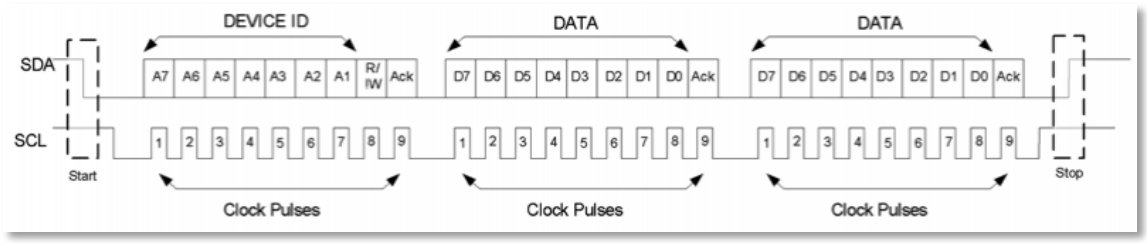

Figure 5: I²C typical message waveform (Digilent)

Bus speed and clock stretching

I²C specification establishes a set of maximum speeds of the SCL line for compliance. The following table shows the standard speeds and their official names:

Name	Speed
Standard mode	100 Kbits/s
Fast mode	400 Kbits/s
Fast mode plus	1000 Kbits/s
HS-mode or High-Speed Mode	Up to 3.4 Mbits/s

Table 2: I²C bus standard speeds

Any I²C compliant device must, at least, be able to communicate at the standard-mode speed. The I²C memory used in this work is compatible with all modes, up to 3.4 Mbits/s.

In the event a slave device is not able to keep up with the clock signal set by the master, it can pull the SCL line down to signal a "clock-stretching" and request the master to withhold the next data bit until the SCL line is released by the slave. Therefore, a master device should test the SCL line before pulling it low to detect such a requirement. This is an optional part of the specification and, in practice, very little slave devices implement the circuitry to drive the SCL line, so clock stretching is not very common.

SPI: Serial Peripheral Interface[12]

SPI (Serial Peripheral Interface) is a serial communication protocol invented by Motorola from 1979 (first appearance) to 1986 (patent) that enables full-duplex communication at moderately high speeds between physically close circuits. It uses four wires, MOSI (Master Out, Slave In), MISO (Master In, Slave Out), SCK (Clock) and CS (Chip Select) and has a single-master, multiple-slave topology.

[12] An official independent SPI specification from Motorola is not available. However, Chapter 6 of the MC68HC11A8 document introduces and describes this protocol:
http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC11A8.pdf

It is, alongside I²C, the de-facto standard for chip communications and it is used, as an example, as the basic protocol for MMC and SD memory cards.

Addressing

All devices share three of the four wires used by SPI: MISO, MOSI and SCK. However, each slave device is selected independently by the master by driving its chip select (CS) line low. Therefore, the total number of wires required corresponds to the following formula: N.Wires = 3 + N.Slaves[13]. Note that, from a device stand point, pins may be called SDI (input) and SDO (output).

By using a dedicated line for each slave addressing, the protocol is greatly simplified, not requiring any address-related message. Considering this protocol overhead reduction and the push-pull driving of the signals, SPI allows for much faster, orders of magnitude greater, communication than I²C. This allows, for instance, for several small (less than 300x300) color LCD screen to use SPI for communication.

Bus read and write operations

SPI transmits data in and out by means of a serial shift register. When a slave is selected, the shift register may be seen as a virtual 16-bit circular shift register, where all outputs shifted out from bit15 are entered into bit0. Note that since the MISO line is push-pull, a master should never pull more than one CS low at the same time when using the standard SPI topology. This could be worked around by using an alternative SPI topology as described later in the chapter.

Figure 6 illustrates the circular behavior of a SPI transmission. A shift occurs when a clock pulse is sent by the master. Serial shifting is, by default, a "shift-left", transmitting the MSB first. Several products also allow LSB first transmissions.

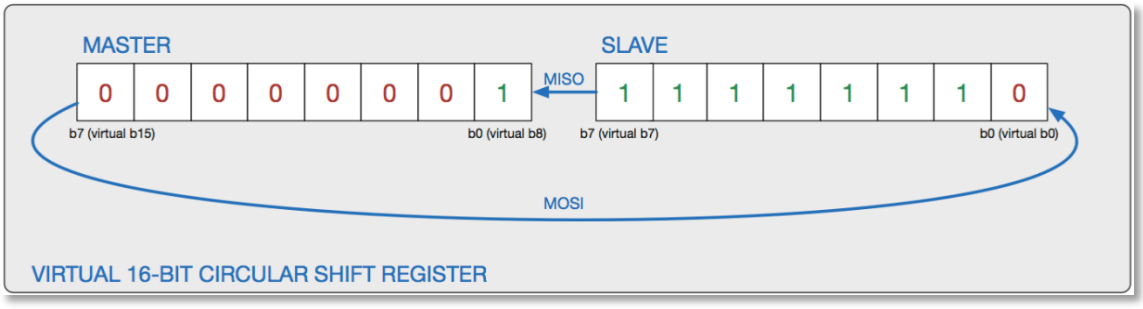

Figure 6: Virtual 16-bit circular shift register of a SPI communication

[13] There are topology variants for SPI configurations that use different number of wires. See "SPI bus topology variants" later in this chapter.

SPI implementations, either hardware or software based, implement a shadow register, or a buffer, for reading and writing the contents of the shift registers, which are not directly accessible. In addition, mechanisms for signaling a complete transmission (8-bit shift) and potential bus collisions (shadow registers are written while a transmission is in progress) are provided for a complete control of the communication.

An overall SPI diagram for a simple master to one slave configuration is shown in *Figure 7*.

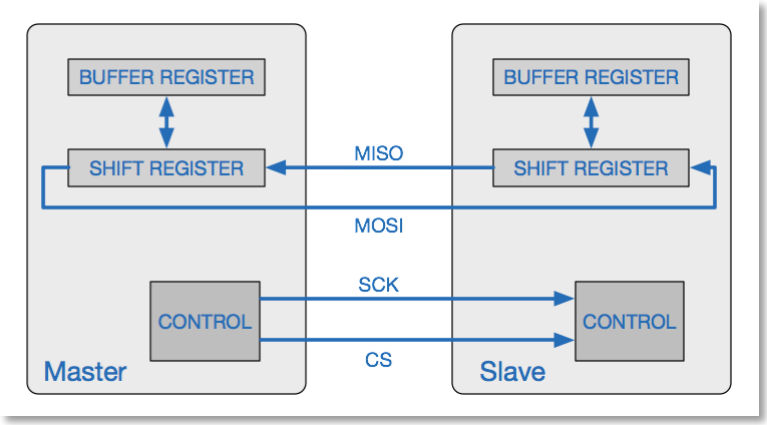

Figure 7: SPI diagram showing internal shift and buffer registers

SPI mode of operation causes a delay of responses from the slave, so a full 16 bits shift is required to get a response to a request (or a command) by the master. A pipeline could be stablished so, when sending multiple data or commands, only an extra shift of 8 bits is required. By default, a 0xFF should be taken as a "null" command or idle bus. This is related to the MISO line as it may be floating (actually it should be pulled-up by an external resistor) when no slave is selected, effectively getting a 0xFF in the MISO line. "Null" commands are often used when reading data from a slave device but, for this purpose, any other slave's non-operational command could be used instead.

Both the master and the slave should also agree on clock polarity and phase, therefore there are four known modes of SPI, Mode 0 to Mode 3. For a master and a slave to be able to communicate, they should meet the following restrictions:

1. **Compatible Speed**. The clock speed is set by the master and should never be higher than the maximum speed supported by the slave.
2. **Same IO voltage levels**. Besides the obvious maximum VCC for both components, 1 and 0 levels should be compliant, so a high-speed CMOS device (HC) may not be compliant with a transistor-transistor logic device (TTL) even if their VCC is the same.
3. **Same SPI mode.** Clock phase and polarity should be the same. Usually, slave devices have a fixed SPI mode, or set of modes, and masters should comply with these limitations.

SPI bus topology variants

SPI uses a standard 3+n wire topology where MOSI, MISO and SCK lines are shared between all devices connected to the bus and each slave is activated by a dedicated "Chip Select" or "Slave Select" signal.

It may be the case that the designer needs to reduce the number of wires used, because of signal or PCB real estate limitations. An alternative bus topology could be used to achieve this goal. However, it comes with a price on speed limitation and/or protocol overhead.
Two common alternative topologies for SPI deployments (namely 3-wire SPI and SPI daisy chain) are described below. Their limitations are also highlighted.

3-wire SPI

3-wire SPI is based on the combination of the MISO and MOSI lines into a single wire. It requires that at least one of the devices participating in the communication supports this mode of operation. A 4-wire SPI device could communicate with a 3-wire SPI device by adding a resistor from the output line to the input line and then using that input line as the data line. This technique sets the output signal in a weakened state against the input signal coming from the other device, therefore, the input line will read the correct value.

The main limitation of this topology is that speed is reduced by half, since it does not allow for pipelining of messages, all shifts need to be 16 bit, the first 8 bit shift the IO line is acting as MOSI and the second shift is acting as MISO. It is a small gain, just one wire, for the number of complications introduced, however it is used sometimes, although not very often. The microcontroller used in this work supports native 3-wire SPI, but we have decided to use a 4-wire topology.

SPI Daisy chain

One of the biggest hardware limitations of SPI is the need for a dedicated "Slave Select" or "Chip Select" signal for each slave. In a circuit with several slaves, it could easily take a lot of the master's IO pins. One common approach to overcome this is using an external demux, therefore requiring only $\log_2$(number of slaves) of pins at the master side. However, if a demux cannot be added, a daisy-chaining of the slaves is possible, creating a virtual shift register of 8 + 8*N.Slaves bits.

This is actually done by connecting all Slave Select lines together and the MISO of the first slave to the MOSI of the second, the MISO of the second to the MOSI of the third, and, generally, the MISO of the N-1th slave to the MOSI of the Nth slave. The MISO of the last slave in the chain is then connected to the MISO of the master, as shown in *Figure 8*.

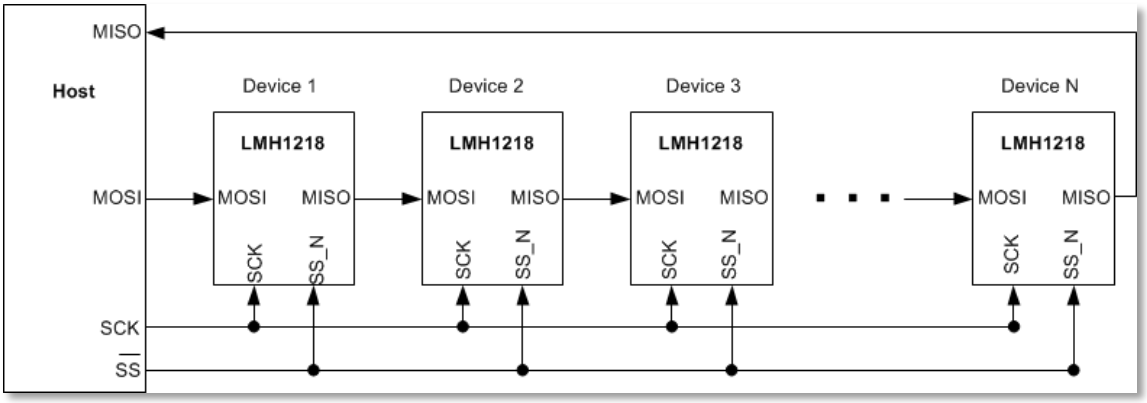

Figure 8: SPI Daisy Chain - Texas Instruments LMH1218 Datasheet

Daisy-chaining a SPI bus offers the advantage of an overall 4-wire topology, but it seriously limits the speed of transmission and introduces a higher level of complexity in the master-slave communication, since the number of 8-bit shifts must correspond to the desired slave recipient in the chain, for sending and to 8*(N.Slaves – slave position) for getting the information back.

Additionally, any slave should ignore any message not intended for it. In a standard topology this is easily achieved by using the CS/SS line, but when in a daisy chain topology, a complex over-the-top protocol must be put in place to create an addressing scheme. Furthermore, most of the SPI slave devices do not support daisy chaining, so this technique is not used very often. Circuits requiring a high number of devices interconnected in the same bus usually implement I^2C instead of SPI.

Asynchronous Serial – UART [14]

UART stands for “Universal Asynchronous Receiver Transmitter” and sets the basics for a one to one, full duplex, asynchronous serial communication. UARTs are usually implemented in hardware in most modern microcontroller and embedded systems. A bitbanging implementation of an UART is quite simple and a VHDL/Verilog definition is also straightforward.

Two devices communicate with each other over a two-line RX (reception) and TX (transmission) pair, allowing for a full-duplex information exchange. Since this is an asynchronous communication, both ends should be configured in advance to use the same speed (clock). Some other details should also be agreed upon such as the shifting direction of the data (LSB or MSB first), inversion of data bits (such as in RS-232 protocol), parity and stop bits.

Summarizing, these are the requirements for two UARTs to communicate:

[14] Asynchronous serial communications date back to the early 20th century, as shown in the patent for a printing telegraph filed in 1908 and granted in 1916:
https://worldwide.espacenet.com/publicationDetails/originalDocument?CC=US&NR=1199011A&KC=A

1. Tolerable voltage ranges and compatible logic levels.
2. Defined speed.
3. Defined shifting direction (MSB, LSB).
4. Defined data polarity (inverted, non-inverted).
5. Defined number of bits to shift (5 to 9).
6. Defined parity signaling (None, even, odd).
7. Defined stop bits (1, 1.5, 2).

Usual configurations are 9600,N,8,1 or 115200,N,8,1 which stand for 9600 (or 115200) bits per second, no parity bit, 8 bits payload and 1 stop bit. Voltage level, shifting direction and data polarity are usually defined in the standard being used. In the TTL UART, voltage level must be TTL compliant, MSB is first and data is not inverted, whereas in the RS-232 standard voltage level is set from -25V to +25V, MSB is first and data is inverted. The RS-232 standard also adds hardware flow by means of RTS (request to send) and CTS (clear to send) signals on dedicated wires, allowing requests to the transmitter to hold any transmission until the receiver is ready.

Figure 9 shows a diagram with a typical UART implementation.

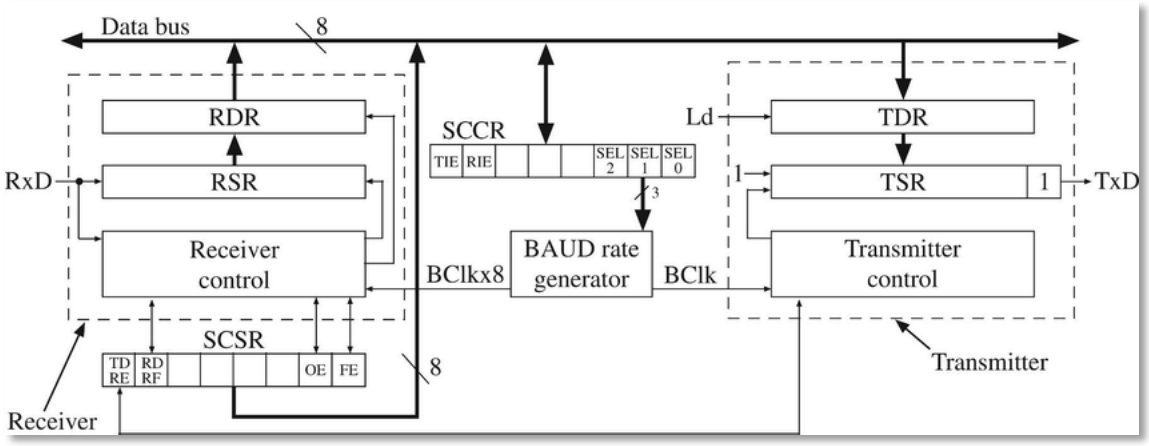

Figure 9: UART Implementation - Umakanta Nanda, 2016 3rd international conference on advance computing and communication systems

Receiver control implements signaling for frame errors (when a received frame is not compliant with the protocol), as well as overrun errors (when a second transmission is received before the previous one is read).

A baudrate generator, usually tied to a timer, is used to shift the data out and sample the incoming data. Shift registers are shadowed by data registers in the same manner explained in the SPI protocol, so they are dedicated registers not directly accessible.

Asynchronous serial waveform

Much like the other protocols used for this work, asynchronous serial is a pure digital protocol that uses one wire for transmitting data and two levels representing a 0 or a 1. In this case, a baud is equal to a bit per second, since there are only two possible states of the line. However, the protocol itself requires an internal state machine to position the participants in the current transmission stage, which is described in detail below.

In an asynchronous serial communication, each data packet is synchronized by a start bit, which is a transition from 1 to 0 after an "idle" or "stop bit" state, indicating that the transmitter is about to begin the payload transmission that would take place at the speed that was agreed between transmitter and receiver. The transition itself is also held for the period corresponding to said speed. Afterwards, the payload is put on the TX line by the transmitter and each bit is held for the specified time. These are the possible states of a pure asynchronous serial transmission:

1. **Idle**: Represented by a logic 1 in the line. It would only transition to "Start bit" on the falling edge of the signal. This state is, by nature, of undefined duration.
2. **Start bit:** Logic 0 in the line, for the speed-defined period (time slot), right after an "Idle" or a "Stop bit" state. It sets the synchronization of the receiver clock for sampling the payload.
3. **Data bits:** Logic 0 or 1 in the line, right after the "Start bit" state, for n time slots, being n between 5 and 9 and set in advance in both the transmitter and the receiver.
4. **Parity bit:** Logic 0 or 1 in the line right after the "Data bits" state, for one time slot. This state is optional and has to be agreed upon in advance by both the transmitter and the receiver.
5. **Stop bit:** Logic 1 in the line right after the "Parity bit" state, if present, or after the "Data bits" state otherwise. It represents the end of the packet transmission. It may have a duration of 1, 1.5 or 2 time slots.

Transitions between states occur automatically after the specified period of time set by the baud rate in all states but from "Idle" to "Start bit". This transition happens on the falling edge of the signal. Receivers usually sample the line in the middle of the time-slot defined by the transmission speed, allowing for line stabilization. After the payload, a parity bit is optionally inserted and then a stop bit (or 1.5 or 2), which is always a logic 1. An idle line is also signaled by a logic 1.

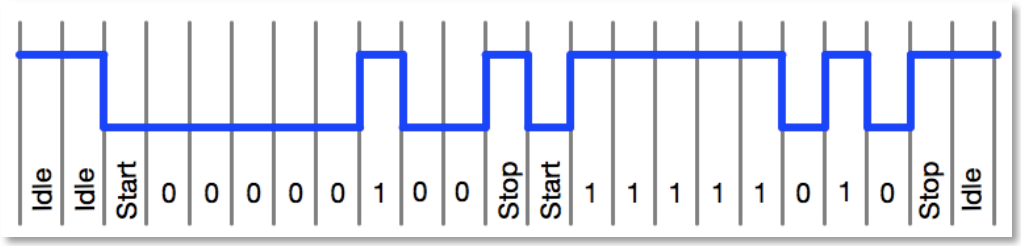

Figure 10: Serial communication of two bytes

Figure 10 shows a two-byte transmission at Speed, N, 8, 1. Payload not inverted and MSB first. The diagram also includes the interpretation for each line state. Note that there is no parity bit since the agreed transmission detail omits it. The data bytes shown in the figure are 0x20 and 0x5F.

In this work environment, Serial is converted to USB CDC (Communication Class Device) by a dedicated ATmega 16u2 microcontroller onboard the Arduino Due. Since USB is a heavy protocol, several chips are available in the market for transforming simple protocols back and forth from one of the standard USB classes. The most common CDC USB chips are the FT230X and the CH340G. In this case, Arduino decided to include a dedicated microcontroller for this purpose. Note that this microcontroller code is fixed and not accessible to the programmer.

CHAPTER 3: RESOURCES AND TOOLS

This chapter summarizes the tools and resources used for building the test platform. It covers the following items:

- Arduino Due platform
- Arduino IDE
- KiCad
- Eclipse Java IDE

Arduino Due[15]

Arduino is an open-source platform, including hardware, libraries, accessories and an integrated development environment for the creation of electronics projects. It uses a simplified version of C++ as programming language and it has become very popular among the DIY (do-it-yourself) community thanks to its simple and integrated approach. It was born as an evolution of a Master´s Thesis from Hernando Barragán on 2003 at the Interaction Design Institute IVREA in Italy.

Arduino Due is an advanced platform based on the ATMEL (Microchip) SAM3X ARM32 microcontroller. Although this microcontroller is quite powerful, it presents serious limitations in the amount of current a pin can source or sink, therefore it is not a very popular platform for driving external hardware.

The most significant features of this platform are:

- ATmega AT91SAM3X8E ARM 32 (Cortex-M3) Microcontroller
- 512KB Flash program memory
- 96KB RAM memory, in two blocks, 64KB and 32KB
- 84 MHz clock
- DMA control (Direct Memory Access)
- 4 serial ports
- 12 8-bit PWM outputs (Pulse Width Modulation)
- SPI peripheral
- CAN peripheral
- 2 I^2C peripherals
- 12 12-bit analog inputs (ADC)
- 2 12-bit DAC (Digital to Analog Converter)
- Arduino Mega compatible form factor

The Figure *11* is an excellent representation of the Arduino Due pinout, ports and capabilities in terms of peripheral and supported current.

[15] https://store.arduino.cc/due

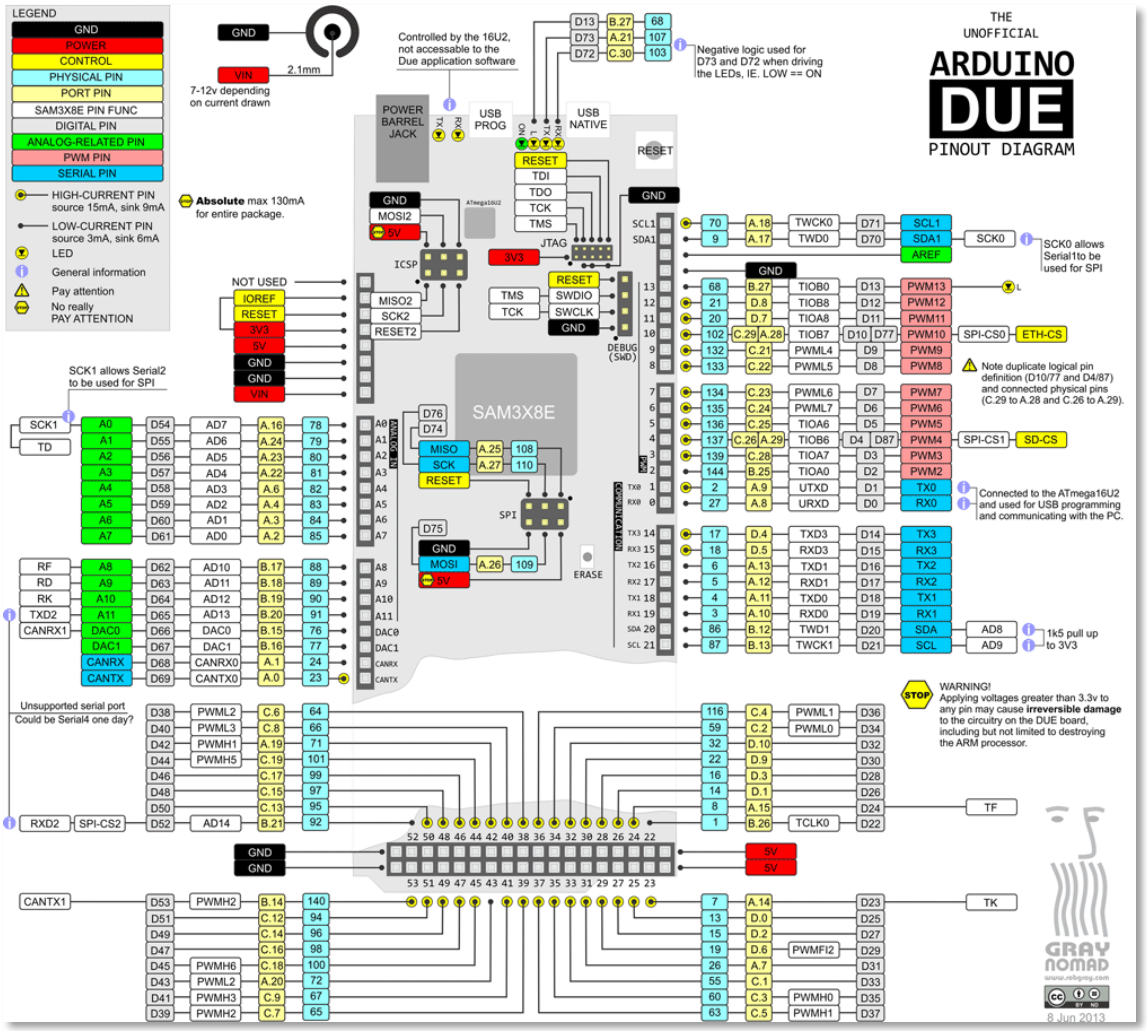

Figure 11: Arduino Due pinout - Rob Gray -https://forum.arduino.cc/index.php?topic=132130.0

Out of the massive peripheral array offered by the Arduino Due, our circuit utilizes a few GPIO pins, SPI, I^{2}C, Serial Port and PWM. *Figure 12* shows the Arduino Due board used in this work.

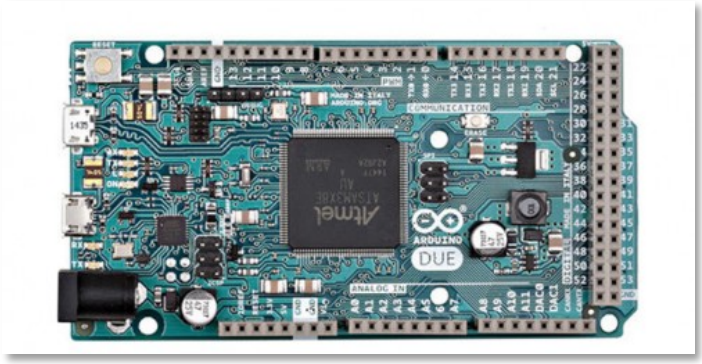

Figure 12: Arduino Due Board. http://www.arduino.cc

Arduino IDE [16]

The Arduino platform offers an open-source integrated development environment (IDE). It is written in Java and it is used to develop, edit and upload programs to Arduino boards and also to some 3rd party circuits. The source-code for the IDE is released under the GPL license, while the C/C++ microcontroller libraries are under the LGPL. Arduino offers two options for using its IDE:

- Arduino Web Editor (Online IDE)
- Desktop IDE (Offline IDE)

Both of them support C and C++ using special rules of code structuring. The IDE supplies a software library, which provides most of the IO procedures used in this work. A basic program is composed by two important functions, one of them (*setup()*) for the setup that is executed at boot and then the main program loop (*loop()*), where the repetitive parts of the program logic should be placed. Both functions are compiled and linked into an executable cyclic executive[17] program. The IDE uses *avrdude* to convert the executable code into a text file in hexadecimal encoding and upload it to the board´s firmware memory.

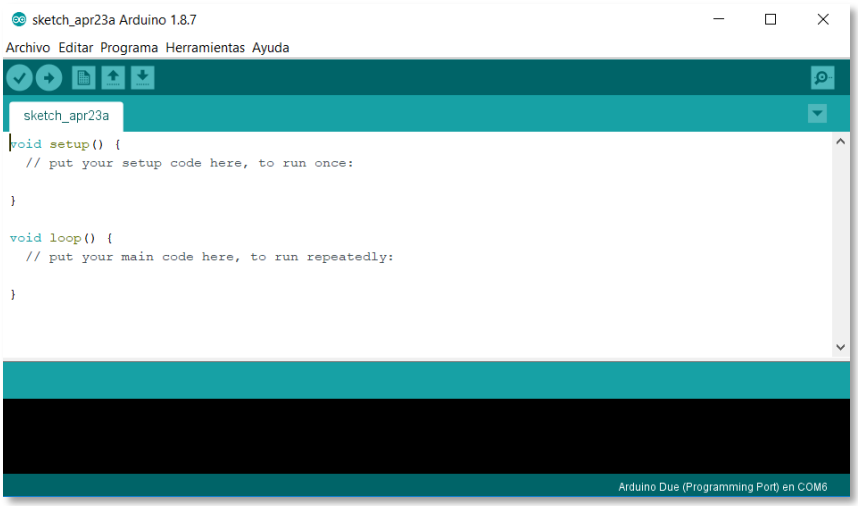

Figure 13: Basic Arduino program sketch – Arduino IDE

The IDE includes example source code for digital and analog communication, control, sensor usage, displays, and sketches/simple programs containing typical use of servos, steppers, etc.

The main user interface presents a very simple layout containing two main panes. At the top, the one for coding and at the bottom, the pane that provides the user with information about program compilation and errors. In order to compile a program, the user must choose the port the Arduino is using to connect to the computer in the first place and, then, click the “next” icon in the upper part to perform the compile operation.

[16] https://www.arduino.cc/en/Main/Software
[17] https://en.wikipedia.org/wiki/Cyclic_executive

Once the program is compiled and loaded into the Arduino it runs in an infinite loop as mentioned above.

To interact via serial port with the board, the IDE offers a very straightforward and simple terminal interface as shown in *Figure 14.*

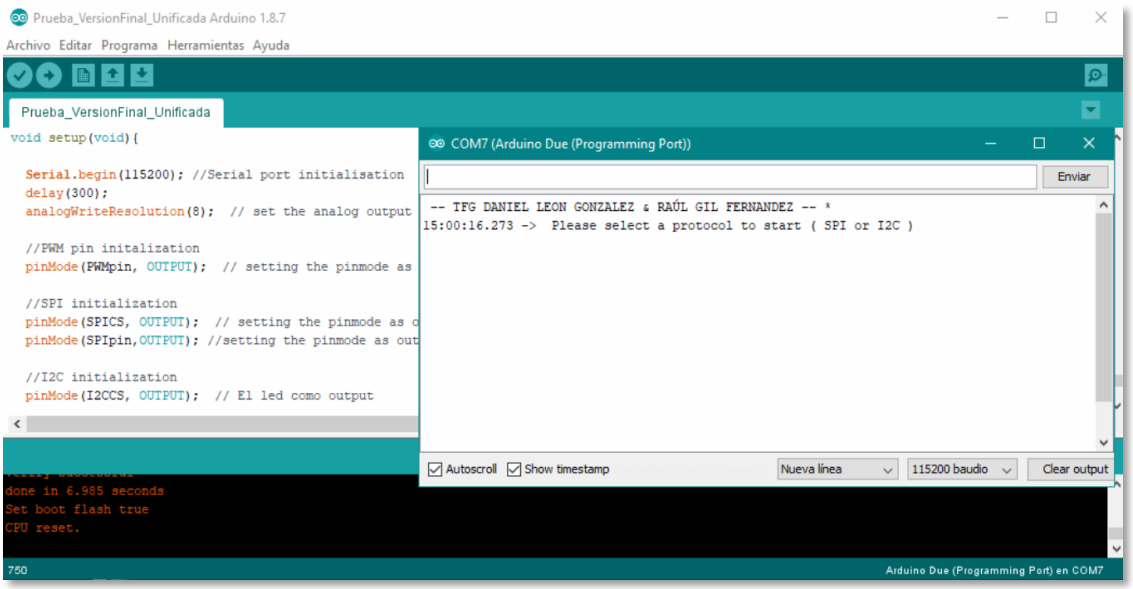

Figure 14: Project Arduino program – Arduino IDE

KiCad[18]

KiCad EDA (Electronic Design Automation) is an open-source software suite that manages the schematic creation, PCB layout design and Gerber tracing for electronic circuit creation. It was originally created in 1992 by Jean-Pierre Charras, but it is now being actively developed and maintained by a community, including CERN, Arduino LLC, Digi-key and the Raspberry Pi foundation. It offers versions for Windows, MacOS and Linux, including ARM-based Linux distributions, so it could run on a Raspberry Pi.

One of the notably missing features of KiCad is the autorouting of signals, however, a third-party program is often used to overcome this limitation. The Java program FreeRouter is usually considered to be part of any KiCad installation.

For our boards, version 5.x of the KiCad EDA suite has been used. Designing a PCB encompasses the four steps detailed bellow:

Schematic capture

Capturing a schematic is done in a subprogram of KiCad called “Eeschema” by placing the used components and creating the connections, either by wires, buses or virtual

[18] http://kicad-pcb.org

wires denoted by labels. All components must first reside in a component library. In *Figure 15*, the variable voltage generator of our main board uses an operational amplifier (or OpAmp) LM358A, which is part of the standard library. Although the KiCad component library is vast, if a component is not included, it can be imported and/or created. At this point, no physical characteristics of the component are considered. Only pin number and its external interface (output, input, bidirectional, power, etc.) describe the components of the schematic.

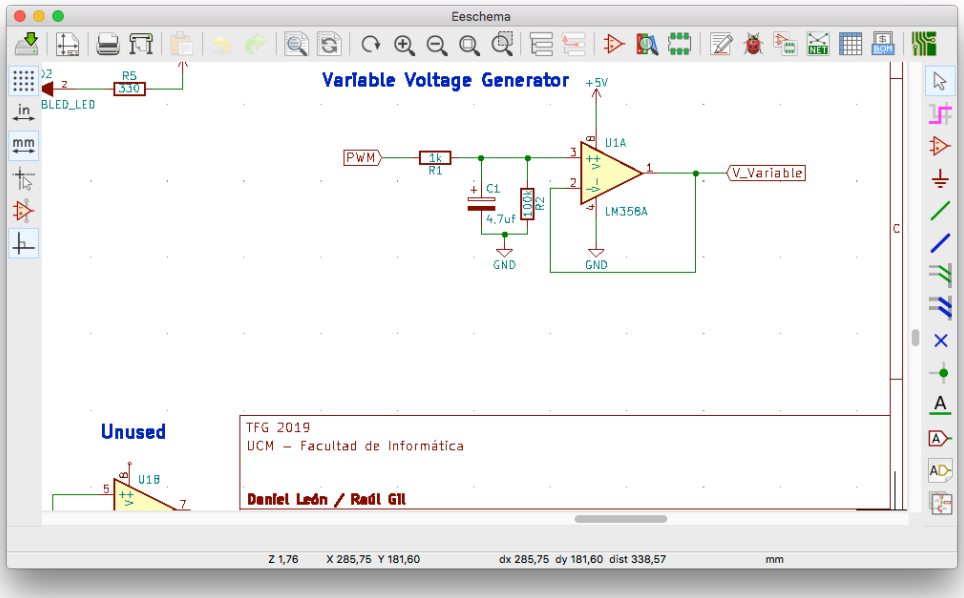

Figure 15: Schematic Capture in KiCad

Once the schematic is finished, KiCad offers an ERC (Electrical Rule Check) option that verifies each pin definition against its connections. This step highlights errors such as short-circuits, non-powered elements, unconnected pins or wrongly connected components, like two output pins connected together. Although using ERC is not mandatory, it is highly advisable.

PCB footprint placement

This task is taken care by a subprogram of KiCad called “Pcbnew”. In a previous step, a “footprint” (which is the physical placement of each component pin, as well as its overall dimensions) is assigned to every schematic element. Again, KiCad provides a wide footprint library and it can be extended or edited to match the designer requirements. As an example, most 74HC circuits will have the same DIP14 footprint. Regardless their schematic functionality is different, pin placement and physical dimensions are the same, therefore they feature the same footprint.

A netlist is then exported from Eeschema to Pcbnew. This contains all components and values, and all connections (usually called “rastnest”).

Then the physical out layer (edge cuts) of the board has to be defined and the component footprints are placed as desired. Other elements such as mounting holes, front or back texts or ground planes can be also added.

Figure 16 shows the aspect of a fully-placed PCB. Note the rastnest in white, showing unconnected pins that have to be connected by each layer's copper tracks.

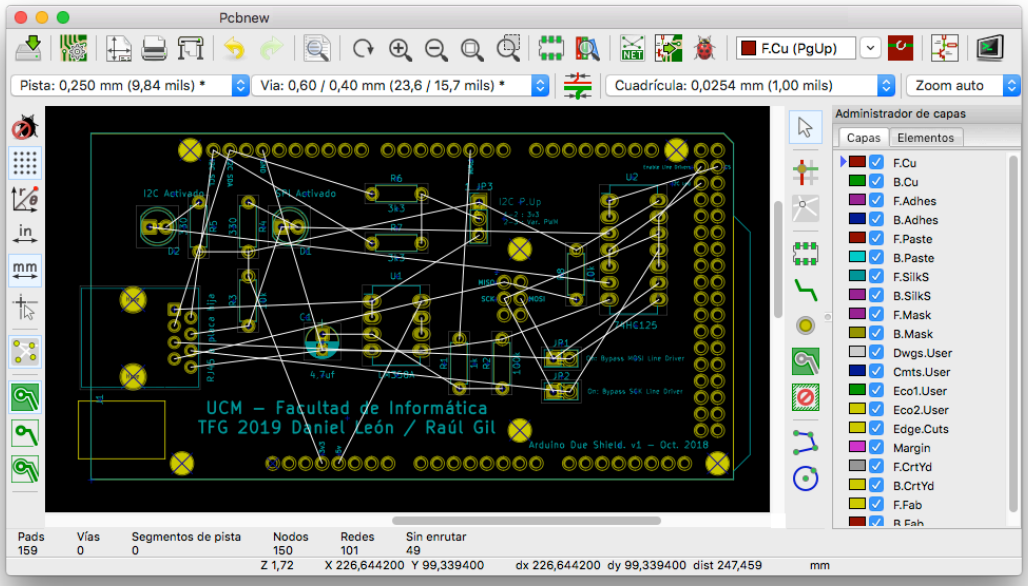

Figure 16: KiCad Pcbnew, components placed, not routed

PCB Track routing

All components in a PCB must be connected according to the schematic. This is done by routing copper tracks between them. Several strategies can be followed, depending on cost and complexity of the PCB. The usual and most economic approach today consists in using a double-layered PCB with tiny conducting holes called "vias". This means that tracks can be placed on both sides of the PCB and those tracks are connected, when needed, by said vias.

KiCad offers the option to manually route the tracks but, unless the board is very basic, this is not really an option. Therefore, another external program called FreeRouter is used. This program imports the PCB, routes the tracks according to specified rules (including via size, track size and separation, etc.) and exports it back to Pcbnew. *Figure 17* shows FreeRouter, which is able to route the main PCB of this work in less than 3 seconds with no vias.

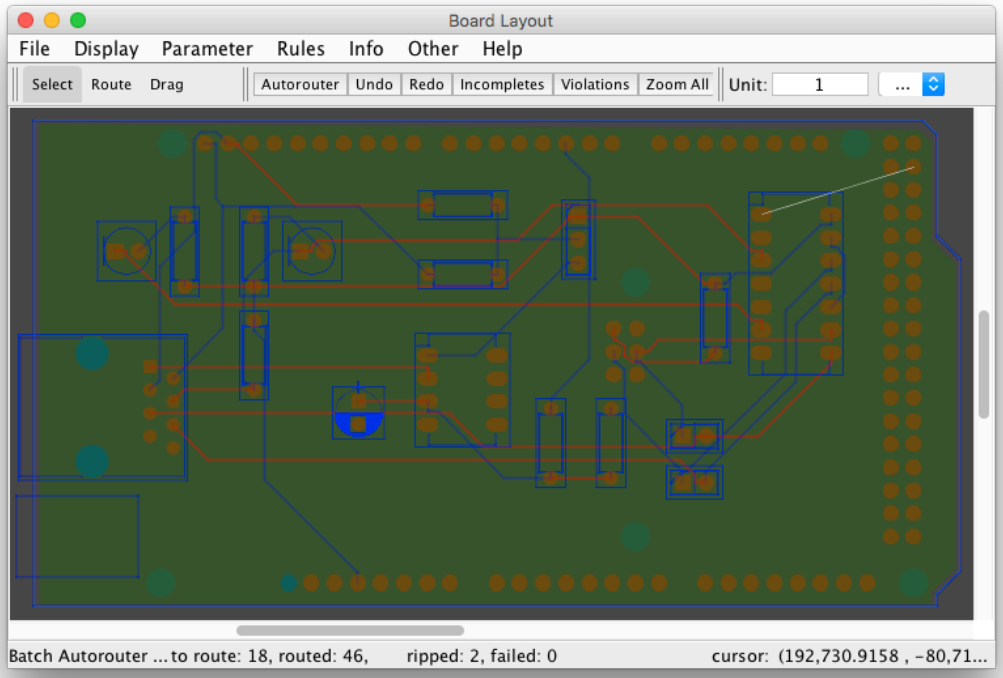

Figure 17: FreeRouter routing the project's Main PCB

Final Checks and Gerber Export

After importing the track routing info back to the PCB, ground planes have to be reconstructed. KiCad also offers a very nice design rule check option that verifies that all physical rules are met. These include unconnected pins, track width, track/vias separation or pad alignment. Even though this is not a mandatory step, it is very important to perform it to ensure a glitch-free PCB generation. *Figure 18* presents the final aspect of the imported board once the ground planes have been reconstructed.

Gerber[19] format is the standard for fabricating PCBs. It includes, in a vector-based approach, the definition for all components of the PCB, edge cuts, texts, drill data, solder mask, copper, mask, etc.

KiCad EDA has an embedded "plotter" for different formats, from EPS to PDF to Gerber itself. Generated Gerber data can then be sent to PCB fabrication houses. There are currently several PCB fabrication brokers for China factories offering very attractive prices at high quality and short lead-times. Our PCBs where sent for production to seeedstudio.io. They took 19 days from order to delivery and the total cost was $22 (around 20€). Other reasonable PCB manufacturing alternatives from China are JLCPCB.com and DirtyPCB.com.

[19] Gerber official website including specification: https://www.ucamco.com/en/gerber

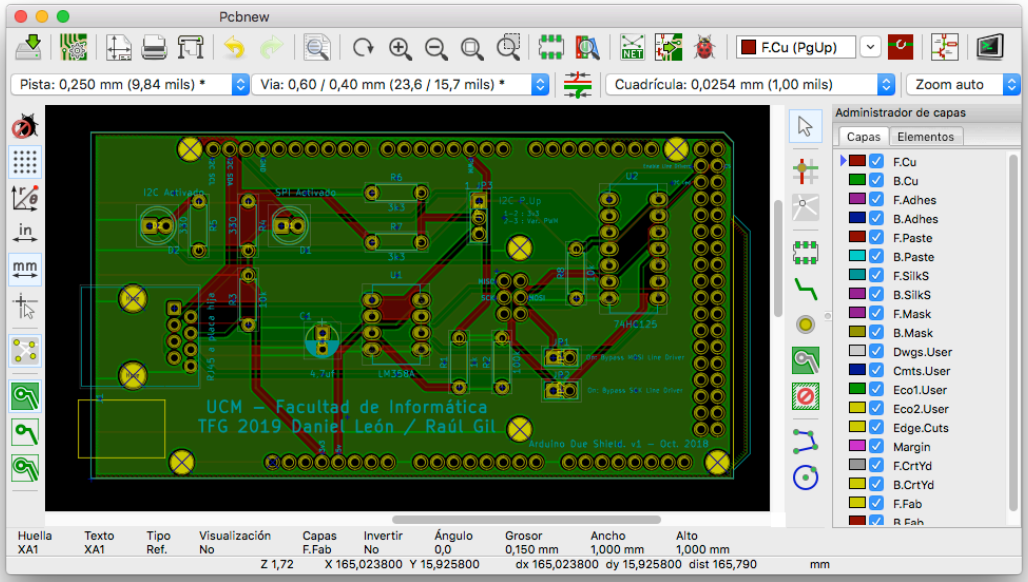

Figure 18: Routed PCB in two layers, including ground planes on both sides

Eclipse IDE for Java [20]

Eclipse is the most widely used Java IDE although it may also be used for C/C++ and PHP languages among several other programming languages. It contains a workspace alongside a plug-in system allowing a very deep environment customization.

Eclipse has its origin in IBM VisualAge, which featured a dual virtual machine for Java and Smalltalk, but as Java popularity and usage began to increase, IBM decided to abandon the dual virtual machine project and create a new platform based just on Java. Eclipse, as we know it today, was born in 2001 with Borland and the Eclipse Foundation, a non-lucrative and open-source project under the Eclipse Public License[21]. This Foundation is currently one of the biggest developing companies worldwide.

Figure 19: Eclipse logo

The software development kit (SDK) provides the user with a mechanism to extend its functionality by using plugins or creating them. This SDK is a free and open-source software released under the Eclipse Public License, but it is incompatible with the GNU General Public License. It is written mostly in Java and it is mainly used for developing

[20] https://www.eclipse.org/ide/
[21] https://www.eclipse.org/legal/epl-2.0/

Java applications, but it may also be used also to create applications in other programming languages via plug-ins, such as C, C++, C#, COBOL, Fortran, Haskell, JavaScript, PHP, Prolog, Python, R, Scala and many others.

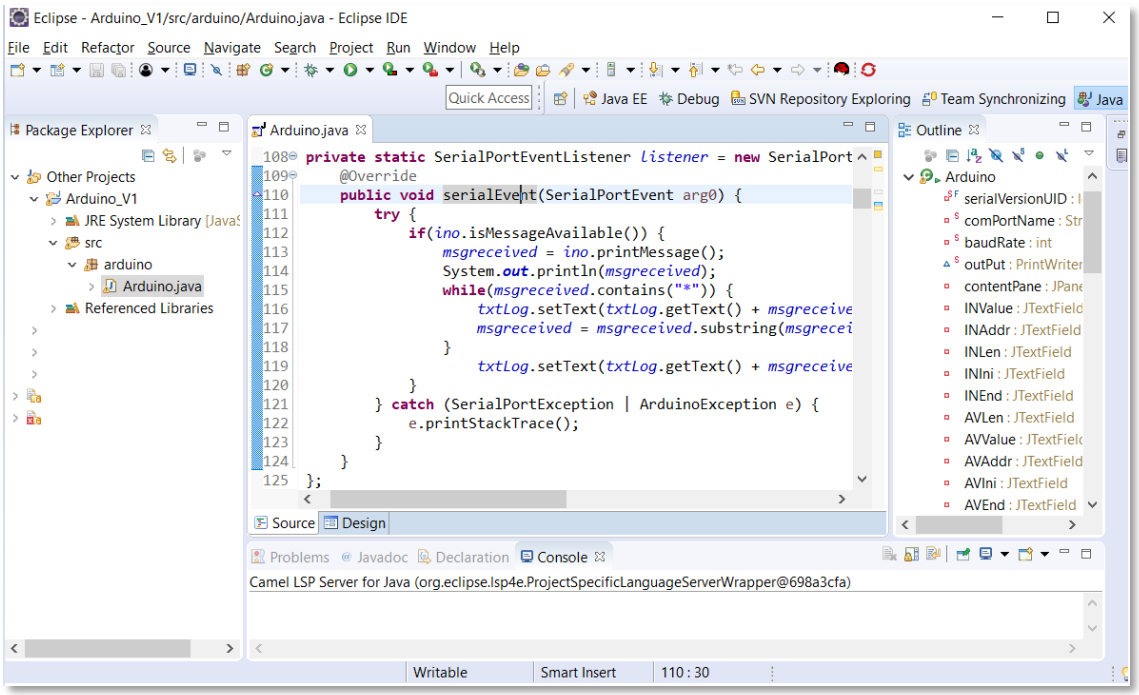

Figure 20: Eclipse interface in developing mode – Eclipse IDE 2018-2019

Figure 20 shows the default pane structure provided by the Eclipse IDE, but it can be modified to satisfy the user's preferences. The main features of this IDE are:

- **Project management:** This IDE provides the user with context-based help and assistance for project creation.
- **Editors and views:** Eclipse interface is focused on a main window formed by different views and preconfigured tabs that makes development easier.
    - Left pane: It shows all existing projects in the actual workspace. Each project is composed by JRE[22] System Library (by default), the code itself and any external libraries referenced/used in the code.
    - Center pane: This is the main view, where the project code is shown. The user may switch between Java classes or interfaces by selecting the upper tabs. The editor does color the different kind of words found in the code (syntax highlighting) for making the code reading easier for the user.
    - Bottom pane: It contains the console logs and it is used for the standard input/output communication while running a program. It also contains a problems tab where errors while compiling or on-execution errors are shown.

[22] Java Runtime Environment

- Left pane: It shows a summary of all the objects and methods involved in the selected class.

- **Code debugger:** It provides a powerful, intuitive and easy-to-use debugger, offering visual feedback and useful information to the developer. The debugger structure is shown in *Figure 21* and contains the following items:
  - Left pane: It shows information about the current execution threads.
  - Center pane: It shows the code. The user can place breakpoints so when the execution arrives to this point, it is stopped, and the code may then be executed step by step.
  - Bottom pane: It contains the console, which is used for the standard input/output communication while running a program and also a problems tab where compiling errors or on-execution errors are shown.
  - Left pane: It contains the current state of the variables in the execution (if stopped) and all the breakpoints placed in the code.
- **Plug-ins collection:** There are a significant amount of them available, made either by the Eclipse Foundation or by third parties. They may be free or not and under different licenses.

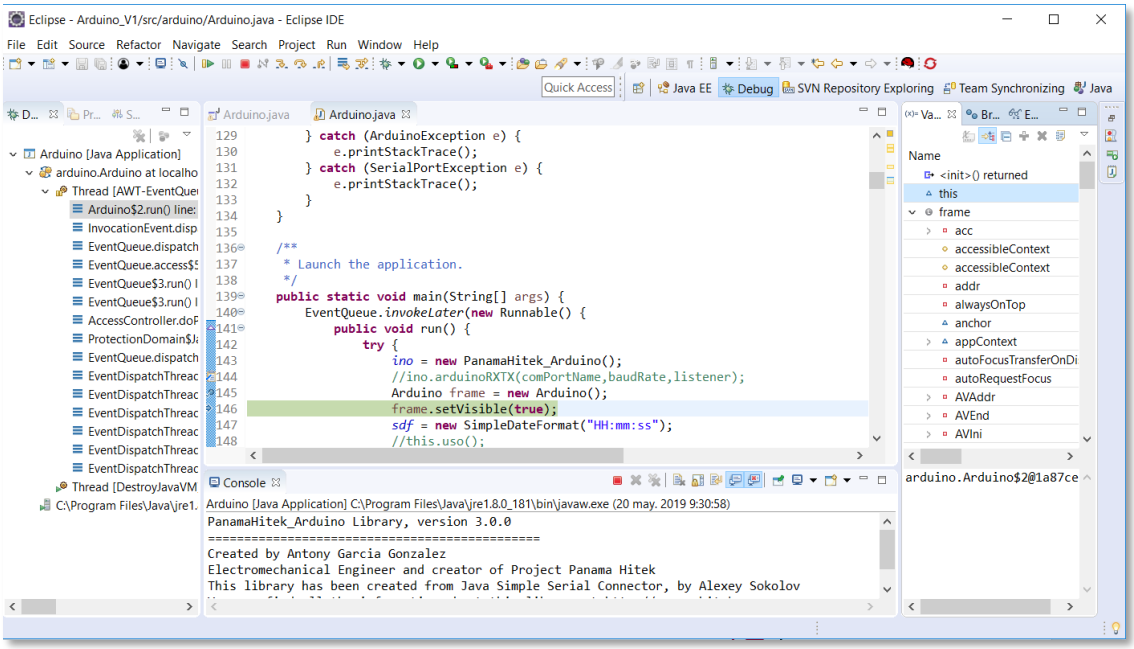

Figure 21: Eclipse interface in debugging mode – Eclipse IDE 2018-2019

For this test platform, we have developed a Java application using Eclipse Java EE IDE for Web Developers (Version: 2018-09 4.9.0) but we could have chosen any other Eclipse IDE.

CHAPTER 4: PROJECT DEVELOPMENT

Having a team of two people and a very well-defined set of goals, we decided to split the tasks as follows:

- Arduino programming (Arduino C++)
  - Implementation of I^2C and SPI memory protocols
  - Implementation of serial listener to respond to PC issued commands
- PC test tool programming (Java)
  - Implementation of graphical interface interacting through serial with the Arduino listener
- Arduino shield board (hardware design)
  - Variable voltage circuit
  - Line Buffers and signaling
  - I^2C pullup selection – operation at lower voltages
- Memory Daughter board (hardware design)
  - Twisted pair and RJ45 connector
  - ZIF sockets

Arduino programming in C++ and PC client programming in Java has been done by Raúl Gil, whereas hardware design, fabrication management and component soldering has been carried out by Daniel León. Raúl used some input from Daniel during the Arduino low-level routines programming, including those related to PWM, serial protocols and line buffer enabling. We decided to use English as the language for the document. Both authors feel comfortable with the language and Daniel performed the overall checking as he holds a C2 CEFR (Proficiency/Mastery) degree in English by the British Council. Directors Juan Antonio Clemente and Juan Carlos Fabero were in charge of the final proofing and corrections and they also hinted appropriate expressions and idioms for a technical report such as this one.

Most of the problems encountered during the development of the project were related to component unavailability. In particular, the I^2C memory could not be supplied by Cypress in the requested SOIC16 packaging, so an alternate packaging (SOIC8-150mil) was acquired from Mouser.com and some patches were applied to the daughter board to accommodate the new part.

Software design. Overall architecture description

The main goal for the software module was to create a program to be run in the Arduino and a PC-based application that communicates with the main board so as to provide a user-friendly interface to the user.

For this purpose, we initially created a pair of applications for each protocol as they are never going to be executed at the same time (either the memory that is under operation works under SPI or I^2C) but we finally mixed them so as to reduce the time to reupload and compile the *.ino* programs to the board. Then, the application is composed by:

- **Arduino program (.ino):** Loaded and run inside the board. This program contains the core functionality. It carries out the communication, following the corresponding protocol, between the mother/master board (Arduino Due) and the daughter board containing the memories. It also controls the standard input/output to manage the execution of commands to the memories. It processes the input strings that the user sends from the PC and executes the related operations.
- **Java program:** The main goal of this Java program is the creation of a visual interface to communicate with the board. With this program, the user no longer needs to write commands to the Arduino input but he or she interacts with the interface instead. This interface connects with the board via serial port and it sends the commands to it as if the user was doing it by using the Arduino IDE serial interface. This makes much easier for the user to handle the memories, since almost everything related to communication is done by the interface.

In order to execute the firmware application, the Arduino program must be uploaded to the board and, once it is running, the Java application must be started on a PC. An overall application workflow is described in *Figure 22*.

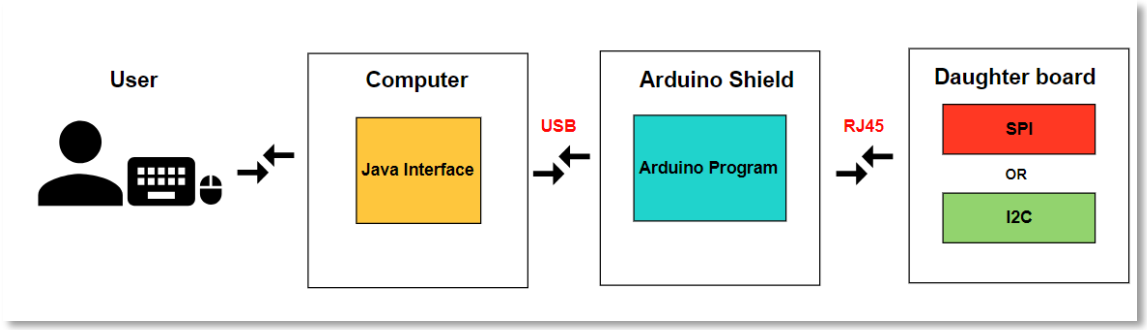

Figure 22: General workflow

Arduino Programming

Implementation of the I²C memory protocol

The Wire[23] library is used for implementing the I²C protocol on the Arduino. It is a standard Arduino library that allows communicating with I²C devices. In our case, the Arduino Due has two I²C peripherals pairs, by default the library uses pins 20 (SDA) and 21 (SCL) but we used the secondary Arduino I²C peripheral ports in our project, so we made use of pins 70 (SDA1) and 71 (SCL1). To make the library work using the non-default ports (SDA1, SCL1) instead of default I²C ports (SDA, SCL) it was necessary to add and change some code in the program:

[23] https://www.arduino.cc/en/Reference/Wire

1. Add the instruction **"extern TwoWire Wire1;"** next to the Wire.h library insertion at the beginning of the code.
2. Instead of calling functions like Wire.xxxx(), use **Wire1.xxxx()**.

Once this was clear, we started coding the Arduino *setup()* function. Inside it, we had to initialize everything needed for the correct operation of the main functionalities, as pointed out above.

The first step for using the protocol is calling the function *Wire1.begin(address)* to initialize the *Wire* library and join the I^2C bus as a master or slave. This function is normally called only once. If an I^2C address is not specified, The Arduino joins the bus as a master. This is exactly the desired bus role for the Arduino board, so an address is not inserted in the *address* parameter.

This is the initial configuration, inserted inside the *setup()* function, required to start working with the protocol. Now the I^2C bus already has 2 components, the nvSRAM CY14B101J memory as a slave and the Arduino playing the master role. Such slave is identified by means of a device address (DA), as it was already pointed out when describing the I^2C protocol. The first byte after a START condition sent to the I^2C bus contains the slave DA of the device which the master intends to communicate with. The format of said address is depicted in *Figure 23*.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	nvSRAM Function Select
1	0	1	0	Device Select ID		A16	R/$\overline{W}$	Selects Memory

CY14X101J Slave Devices

Memory, 128K × 8

Figure 23: Slave device addressing – Cypress - http://www.cypress.com/files/cy14c101jcy14b101jcy14e101j-1-mbit-128k-x-8-serial-i2c-nvsrampdf

As it can be seen, bits 7-4 are fixed, whereas bits 3-2 (Device Select ID) must be set up manually. In this work, the I^2C bus only has 1 SRAM, so both of them have been physically tied to a constant value, 0 (these are pins A1 and A2 of the chip, see *Figure* 24). Bit 1 is the 16^{th} bit of the word address (WA) that will be read from or written to; whereas bit 0 must be set to 0 when writing or 1 for reading. Thus, following this format, in our program, the DA will take the following values:

- 1010 00 A16 1 -> Reading
- 1010 00 A16 0 -> Writing

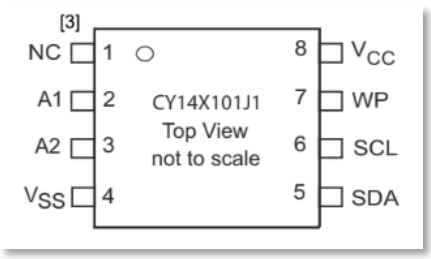

Figure 24: 8-pin SOIC I2C nvSRAM pinout Cypress - http://www.cypress.com/files/cy14c101jcy14b101jcy14e101j-1-mbit-128k-x-8-serial-i2c-nvsrampdf

As mentioned above, the A16 bit refers to the MSB of the WA. The memory has a 128K x 8 bits size, so in order to address it, 17 bits (3 bytes) would be required for WA. The method used in this memory to save an extra byte for in WA is mapping the A16 bit in bit 1 of DA (see *Figure 23*).

Summarizing, DA can take the following 4 possible values:

1. (BIN) 1010 00 0 1 // (DEC) 161 -> A16 = 0, Write
2. (BIN) 1010 00 1 1 // (DEC) 163 -> A16 = 1, Write
3. (BIN) 1010 00 0 0 // (DEC) 160 -> A16 = 0, Read
4. (BIN) 1010 00 1 0 // (DEC) 162 -> A16 = 1, Read

This is the logic based on the memory internal structure but, as explained below, while using Arduino Wire library, these values are not exactly the previously shown.

WRITE OPERATION

The procedure to perform a *write* followed by the memory (and also exposed in its datasheet) is described in *Figure 25.*

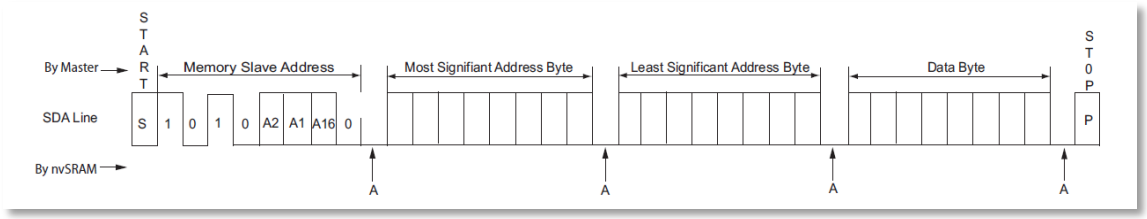

Figure 25: nvSRAM I²C Write operation – Cypress - http://www.cypress.com/files/cy14c101jcy14b101jcy14e101j-1-mbit-128k-x-8-serial-i2c-nvsrampdf

Arduino Wire library functions are used to perform a data transmission to the memory:

1. *Wire.beginTransmission*[24]*(address):* It initializes a transmission to the I²C slave device with the given DA ("Memory Slave Address" in *Figure 25*).
2. *Wire.write(byte):* It sends the specified *byte* through the I²C bus. For a write operation to be successful, the "Most Significant Address Byte", the "Least Significant Address Byte" and the "Data Byte" (or bytes) must be sent (see *Figure 25*). Hence, this function must be called three or more consecutive times.
3. *Wire.endTransmission*[25]*(stop):* It ends the transmission initiated by *beginTransmission()* and sends the buffered data to the memory. The *stop* argument is a Boolean value. If this argument is *false*, it sends a restart condition. This is used in read operations, as explained below.

[24] https://www.arduino.cc/en/Reference/WireBeginTransmission
[25] https://www.arduino.cc/en/Reference/WireEndTransmission

As mentioned above, the DA is a byte, but the *beginTransmission(address)* function only accepts an argument of 7 bits. Using the function *endTransmission()* always adds the final bit (bit 0 in *Figure 23*), with value 0, to the *address* argument. So, we can say that every transmission performed using these two functions is a "write" operation as this final bit determines it.

Therefore, the *address* argument would take the following values. Bits written in bold indicate different values for A16:

- (BIN)1010 00 **0** // (DEC) 80
- (BIN)1010 00 **1** // (DEC) 81

Figure 26 shows an example of write operation in an I²C memory.

```
void write(uint32_t address, uint8_t data_byte) {


  uint8_t value;
  uint8_t addr_xhi;
  uint8_t addr_lo;
  uint8_t A16;
  int deviceAdd;

  A16 = (address >> 16) && 0x01; // Bit A16
  addr_xhi=(address >> 8);// && 0xFF; // Bits A8-A15
  addr_lo=(address & 0xFF); // Bits A0-A7

  if(A16 == 0) deviceAdd = 80;
  else if (A16 == 1) deviceAdd = 81;
  else{
    Serial.println("ERROR, Invalid device address");
    return;
    }

  Wire1.beginTransmission(deviceAdd);
  Wire1.write(addr_xhi);
  Wire1.write(addr_lo);
  Wire1.write(data_byte);
  if(Wire1.endTransmission() == 0) Serial.print(" -> OK <- ");
  else Serial.print(" -> ERROR <-");
  Serial.print( " Write Address: ");
  Serial.print(address);
  Serial.print( " Value: ");
  Serial.println(data_byte);

}
```

Figure 26: Basic code scheme to perform a write operation in an I²C nvSRAM

READ OPERATION

The procedure to perform a *read*, as included in the memory´s datasheet is shown in *Figure 27.*

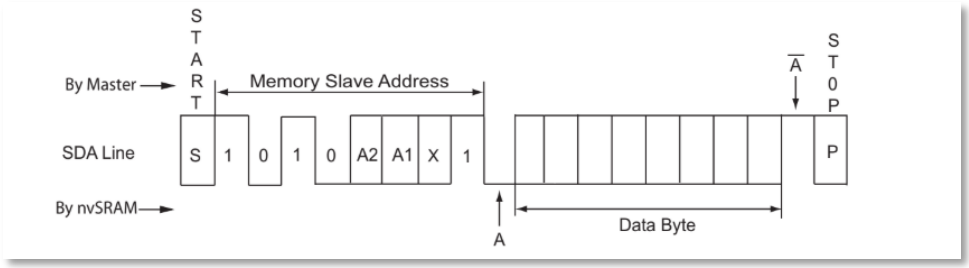

Figure 27: I²C nvSRAM Read operation carried out in the position indicated by the internal counter– Cypress - http://www.cypress.com/files/cy14c101jcy14b101jcy14e101j-1-mbit-128k-x-8-serial-i2c-nvsrampdf

The internal operation of the memory for reading must be considered, as it has a strong influence over the software modelling. The memory has an address counter, and if a read operation is performed as in *Figure 27*, the data byte returned by the memory would be the one pointed by this address counter. After returning it, the counter increments its current value in one position.

Bearing this behavior in mind, if a random address read is required, the address counter has to be set to the desired position first, and then a simple read operation must be performed following the scheme of *Figure 27*. A read operation carried out in any memory position of the memory is described in *Figure 28*.

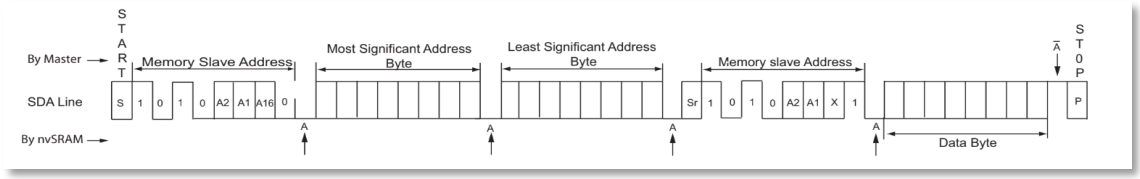

Figure 28: I²C nvSRAM Read operation carried out in a position indicated by a given WA. WA is reconstructed by using "Most Significant Byte", "Least Significant Byte" and A16 – Cypress - http://www.cypress.com/files/cy14c101jcy14b101jcy14e101j-1-mbit-128k-x-8-serial-i2c-nvsrampdf

This operation is actually formed by a write operation in the SRAM with no data and finalized with a "repeated start", which is followed by a simple read operation. Thus, the sequence of functions that the Arduino must invoke is the following:

1. *Wire.beginTransmission*[26]*(address):* It has the same purpose as in a write operation.
2. *Wire.write (byte):* Note that, this time, only two bytes are transmitted ("Most Significant Byte" and "Least Significant Byte"). No data is transmitted this time.

[26] https://www.arduino.cc/en/Reference/WireBeginTransmission

3. *Wire.endTransmission*[27]*(stop):* This time, the value of *stop* must be *false* to indicate the existence of a "Repeated Start". This is shown in *Figure 28* with the bit labeled as "Sr".
4. Read operation as indicated in *Figure 27*. *Wire.requestFrom()* and *Wire.read()* functions are invoked to this end.

The only way to move the address counter is while writing, since reading as indicated in *Figure 27* is always performed on the current address counter position. This is the reason why, in order to carry out a reading in any position, a "dummy" write operation is made in the first place (just to modify the counter), followed by the actual read operation.

Now, let us explain a little bit in detail Steps 2 and 4:

In Step 2, the address to be read (i.e. WA) must be inserted between *Wire.beginTransmission (address) and Wire.endTransmission (stop)* functions, so the *Wire.write(data)*[28] should be used twice to send, first MSBs (A15-A8) and then LSBs (A7-A0) of the WA. No payload data should be sent yet. *Wire.write(data)* function does not actually write anything to the bus. Instead, it just inserts the argument *data* inside the data buffer whose contents are going to be sent to the memory when *endTransmission()* is called. This performs an ***"empty write"***, having the solely goal to move the address counter to the position we want to read.

In Step 4, data are requested. The Wire library has a specific function to request data to the memory called *Wire.requestFrom(address, quantity)* [29]*,* accepting two arguments:

- *address*: the same argument *beginTransmission()* function uses (i.e., the 7 MSBs of the DA). The missing bit is automatically set to 1 by the function, indicating that a read operation is performed. Thus, the values used for this *address* argument are (A16 bit in bold):
  - (BIN) 1010 00 **0** // (DEC) 80
  - (BIN) 1010 00 **1** // (DEC) 81
- *quantity*: requested bytes to read from the memory (starting from the address counter location)

All requested data is stored in a buffer. The maximum number requested bytes is 32, which is the size of the buffer used by this library. After requesting bytes from the memory, we should check for data availability in the buffer by calling the *Wire.available()* function. It returns the number of bytes available that should be equal to the *quantity* argument value used in *requestFrom()* function. Finally, we retrieve the data from the buffer using *Wire.read()* function.

[27] https://www.arduino.cc/en/Reference/WireEndTransmission
[28] https://www.arduino.cc/en/Reference/WireWrite
[29] https://www.arduino.cc/en/Reference/WireRequestFrom

Figure 29 shows an example of the code for a read operation in an I²C memory.

```
uint8_t read(uint32_t address) {

  uint8_t value;
  uint8_t addr_xhi;
  uint8_t addr_lo;
  uint8_t A16;
  int deviceAdd;

  A16 = (address >> 16) && 0x01; // Bit A16
  addr_xhi=(address >> 8);// && 0xFF; // Bits A8-A15
  addr_lo=(address & 0xFF); // Bits A0-A7

  if(A16 == 0) deviceAdd = 80;
  else if (A16 == 1) deviceAdd = 81;
  else{
    Serial.println("ERROR, Invalid device address");
    return -1;
    }

  // An "empty write" is performed to move the address counter
  // to the desired position
  Wire1.beginTransmission(deviceAdd);
  Wire1.write(addr_xhi);
  Wire1.write(addr_lo);
  Wire1.endTransmission(false);

  // Request the data from the memory
  Wire1.requestFrom(deviceAdd,1);

  // (Optional) Make sure the data is received.
  if(Wire1.available() == 0) Serial.println( "Not available");
  else{
        //Retrieve the data from the buffer
        value = Wire1.read();
        Serial.print( " -> OK <- Read Address: ");
        Serial.print(address);
        Serial.print( " Value: ");
        Serial.println(value);
        Serial.print("*");
  }

  return value;
}
```

Figure 29: Basic code scheme to perform a read operation in an I²C nvSRAM

Implementation of SPI memory protocol

To implement the SPI protocol, we made use of the SPI[30] Arduino standard library. It allows communicating with devices using said serial protocol. In our case, the Arduino Due has a set of six default pins for SPI protocol. In this project we use the main three of them (SPI_MISO, SPI_MOSI and SPI_SCK) via the previously mentioned library.

The first step is configuring the *setup()* function inside our code. First, we configure the "SPI pin" (in our case, pin 22 of the Arduino Due) as output and set it to low which enables the output of the line buffers for SCK and MOSI. After that, we should start the protocol. This is done by the *SPI.beginTransaction()* function that accepts an argument that defines the SPI settings. This argument is an object that is composed by 3 parameters (speed, endianness and mode):

- *speedMaximum*: The maximum speed of communication. For a SPI chip rated up to 20 MHz, use 20000000. We set it to 4000000 for a 4MHz shift clock.
- *dataOrder*: MSBFIRST or LSBFIRST. Set to MSBFIRST.
- *dataMode* : SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3. This particular memory supports modes 0 and 3. Mode 3 is used in our setup.

SPI.beginTransaction() initializes the SPI bus. There is also another pin used in every operation, the chip select pin (in our case, pin 23 of the Arduino Due). It is used for selecting a slave device. The master must pull it down (active low) before performing any operation. Otherwise, any data sent through MOSI is ignored by the slave.

Every desired transaction must be encapsulated between functions *SPI.begin()* and *SPI.end()*.

- *SPI.begin():* Initializes the SPI bus by setting SCK, MOSI and SS to outputs, pulling SCK and MOSI to low and SS to high.
- *SPI.end():* Disables the SPI bus (leaving pin modes unchanged).

In between this encapsulation, we must transmit all the information required for performing the desired operation to the selected memory. This is done by using the *SPI.transfer()* function. SPI is a full-duplex protocol, so calling this function will send and receive a value at the same time. This function is used as a mechanism to perform both read and write operations.

OPERATION CODES

The operation code is the first byte transmitted to a memory just after setting its chip select pin to low. *Figure 30* shows the existing operation codes. The next subsections will refer to them when describing the read and write operations on SPI memories.

[30] https://www.arduino.cc/en/Reference/SPI

Name	Description	Opcode
WREN	Set write enable latch	0000 0110b
WRDI	Reset write enable latch	0000 0100b
RDSR	Read Status Register	0000 0101b
WRSR	Write Status Register	0000 0001b
READ	Read memory data	0000 0011b
FSTRD	Fast read memory data	0000 1011b
WRITE	Write memory data	0000 0010b
SLEEP	Enter sleep mode	1011 1001b
RDID	Read device ID	1001 1111b
Reserved	Reserved	1100 0011b
		1100 0010b
		0101 1010b
		0101 1011b

Figure 30: CY15B102Q and CY15B104Q SPI opcodes– Cypress - http://www.cypress.com/file/209146/download

READ OPERATION

The procedure to perform a *read* followed by the memory, as exposed in its datasheet, is shown in *Figure 31*.

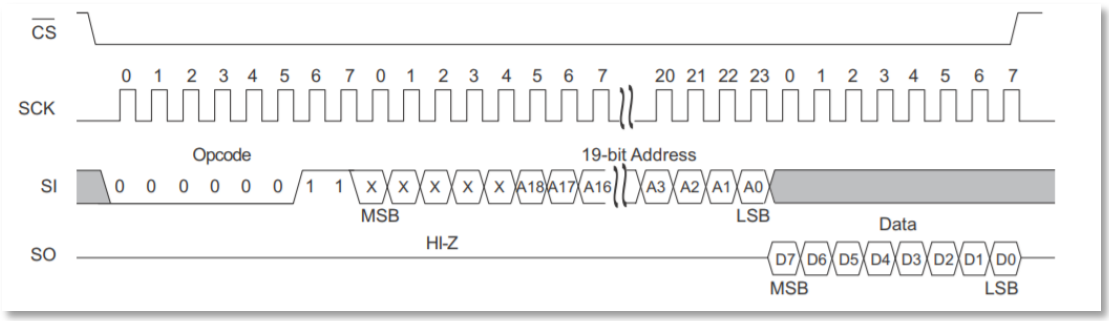

Figure 31: CY15B102Q and CY15B104Q SPI READ operation – Cypress - http://www.cypress.com/file/209146/download

Following the READ opcode (0000011), a 19-bit address (A18-A0) must be sent, as three bytes are required to access any location for this 4-Mbit memory (512K x 8).

The operation flow is the following:

1. Select the target memory chip by pulling down the chip select pin (CS).
2. Send the operation code (READ) and the desired memory address to read from.
3. Perform an "empty transfer" in order to obtain the data returned by the memory. The *data* sent by the *SPI.transfer(data)* function is not relevant since the SI input is ignored by the slave when reading data bytes.
4. Finally, as the read is finished, the chip select pin must be changed back to high.

Reads can be performed in burst mode. If CS is held low, the device would automatically return the next address value after each byte of data is delivered.

The code for performing a read operation in shown in *Figure 32.*

```
uint8_t read(uint32_t address) {

  uint8_t value;
  uint8_t addr_xhi;
  uint16_t addr_lo;
  addr_xhi=(address >> 16);
  addr_lo=(address & 0xffff);
  writeEnable();

  SPI.begin();
  digitalWrite(CS,LOW);
  SPI.transfer(READ);
  SPI.transfer(addr_xhi);
  SPI.transfer16(addr_lo);
  value=SPI.transfer(0xAA);
  Serial.print( " -> OK <- Read Address: ");
  Serial.print(address);
  Serial.print( " Value: ");
  Serial.println(value);
  digitalWrite(CS,HIGH);
  SPI.end();

  return value;
}
```

Figure 32: Basic code scheme to perform a read operation using SPI

WRITE OPERATION

The procedure to perform a *write* followed by the memory, as exposed in its datasheet, is represented in *Figure 33.*

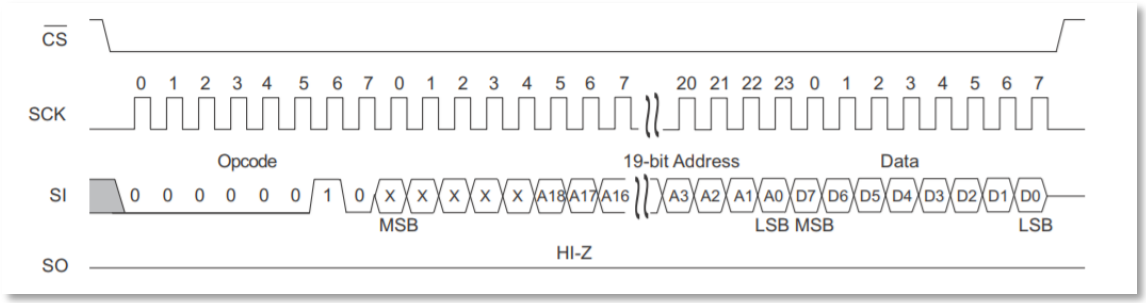

Figure 33: CY15B102Q and CY15B104Q SPI WRITE operation – Cypress - http://www.cypress.com/file/209146/download

Before performing any write operation, we must enable writes to the memory. To understand this concept, we should take a look at the memory chip Status Register. This register contains memory status values. The meaning of this register is described in *Figure 34* and *Figure 35.*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPEN (0)	X (1)	X (0)	X (0)	BP1 (0)	BP0 (0)	WEL (0)	X (0)

Figure 34: CY15B102Q and CY15B104Q status register. Default value is 64 – Cypress - http://www.cypress.com/file/209146/download

Bit	Definition	Description
Bit 0	Don't care	This bit is non-writable and always returns '0' upon read.
Bit 1 (WEL)	Write Enable	WEL indicates if the device is write enabled. This bit defaults to '0' (disabled) on power-up. WEL = '1' --> Write enabled WEL = '0' --> Write disabled
Bit 2 (BP0)	Block Protect bit '0'	Used for block protection. For details, see Table 4.
Bit 3 (BP1)	Block Protect bit '1'	Used for block protection. For details, see Table 4.
Bit 4-5	Don't care	These bits are non-writable and always return '0' upon read.
Bit 6	Don't care	This bit is non-writable and always returns '1' upon read.
Bit 7 (WPEN)	Write Protect Enable bit	Used to enable the function of Write Protect Pin ($\overline{WP}$). For details, see Table 5.

Figure 35: CY15B102Q and CY15B104Q status register, bit definitions – Cypress - http://www.cypress.com/file/209146/download

Using the RDSR operation code (*Figure 30 and Figure 36*) the bus master may check the Status Register of the memory.

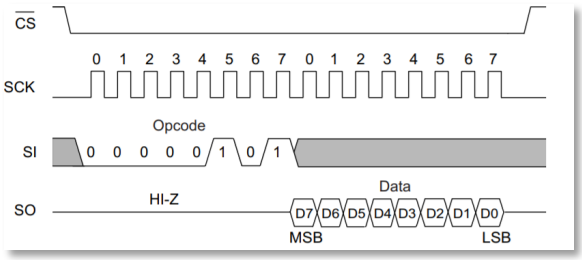

Figure 36: CY15B102Q and CY15B104Q SPI RDSR operation– Cypress - http://www.cypress.com/file/209146/download

In order to make a write operation, the Write Enable bit of the status register (bit 1 (WEL), *Figure 34* and *Figure 35*) must be set to 1. Since its default value is 0, we should change the value of that bit from 0 to 1 each time a write operation is to be performed. To this end, we have created an auxiliary function called *writeEnable()* (*Figure 37*).

```
void writeEnable(){
  SPI.begin();
  digitalWrite(CS,LOW);
  SPI.transfer(WREN);
  sr=SPI.transfer(0x00);
  digitalWrite(CS,HIGH);
  SPI.end();
  }
```

Figure 37: writeEnable() function

It must be considered that the WEL bit is automatically cleared when CS is set to high after a WRDI, WRSR or a WRITE operation. So, it must be enabled before every single write operation.

After writes are enabled, a WRITE opcode, followed by a three-byte address containing the 19-bit address (A18-A0) of the first data byte to be written must be sent. Next, data bytes are written sequentially. The addresses are incremented internally as long as CS is low. The basic code for this operation is shown in *Figure 38*.

```
void write(uint32_t address, uint8_t data_byte) {

  uint8_t value;
  uint8_t addr_xhi;
  uint16_t addr_lo;
  addr_xhi=(address >> 16);
  addr_lo=(address & 0xffff);
  writeEnable();

  SPI.begin();
  digitalWrite(CS,LOW);
  SPI.transfer(WRITE);
  SPI.transfer(addr_xhi);
  SPI.transfer16(addr_lo);
  SPI.transfer(data_byte);
  Serial.print( " -> OK <- Write Address: ");
  Serial.print(address);
  Serial.print( " Value: ");
  Serial.println(data_byte);

  digitalWrite(CS,HIGH);
  SPI.end();
}
```

Figure 38: Basic code scheme to perform a write operation using SPI

Implementation of serial listener to respond to PC issued commands

The Arduino board and the user´s computer communicate by means of an USB port implementing an asynchronous serial protocol (CDC[31] profile). It is configured in the *setup()* function using *Serial.begin(baudrate).* Once this is done, functions *Serial.read()* and *Serial.write(val)* may be used. Note that parameter *val* is a single byte value.

The source *.ino* program handles the strings introduced by the user using serial communication. Being inside the *loop()* function (*Figure 39*), it reads the serial input and calls the function *procesa()* to process it.

The procesa() function acts like an organizer and, depending on the input string content, it calls an auxiliary function that executes the desired functionality. Inside these intermediate functions, despite the processing differences, all of them use the base functions *read()* (*Figure 32*) and *write()* (*Figure 38*) for SPI, and *read()* (*Figure 29*) and *write()* (*Figure 26*) for I^2C as explained in the previous sections.

[31] CDC: Communication Device Class

```
void loop(){
  String cadena;

  bool leido=false;
  while (!leido) {
    if (Serial.available()>0) {
      cadena=Serial.readStringUntil(10); // LineFeed
      cadena.toUpperCase();
      leido=true;
    }
  }
  procesa(cadena);

  }
```

Figure 39: Dealing with serial input inside the loop() function

The functions the user can call are the following:

- WA <value> <addr>: write <value> to address <addr>
- WR <value> <len>: write <value> for <len> bytes from 0x0000
- WP <ini> <end> <value>: write <value> from address <ini> to <end>
- ALLW <value> : write <value> to all memory addresses
- RA <addr>: read value at address <addr>
- RD <len> : read <len> bytes from 0x0000
- RP <ini> <end>: read from address <ini> to <end>
- ALLR : read all memory addresses
- PWM <value>: Change the duty cycle of the PWM to <value>
- SR: read StatusRegister (Just in SPI)

PC Test Tool Programming

Arduino IDE features a built-in serial monitor to communicate with Arduino board but, in our case, we have developed a Java Swing graphical interface to take care of this task.

This GUI acts as a bridge between the user and the board. The Panama Hitek Arduino[32] library that uses Java Simple Serial Connector is used for this purpose.

This GUI has to invoke the *arduinoRXTX(portname, baud rate, listener)* method. This method initializes the connection between Java and Arduino and allows us to either send or receive data. About the parameters:

- *portname*: The port used in the connection. In our case we are using port COM7.
- *baud rate*: Number of signal units per second. It has to be the same as the one declared on the *.ino* program.

[32] http://panamahitek.com/libreria-panamahitek_arduino/

- *listener:* A serial port event listener in charge of dealing with output messages from the *.ino* program.

The listener receives events every time the Arduino board generates some data that has to be displayed. It takes the message and prints it on the GUI logs. The listener acts as a bridge between the Arduino and the user, hence it does not process any data.

For sending data to the Arduino Due, we created the function *sendDataToArduino(data).* It executes the *Panama Hitek* library function *sendData(message)* that sends a string variable via serial port. This string will be received by the board, passed by to *procesa()* and hence processed by the main board.

The GUI (*Figure 40*) contains 4 main panes:

1. Functions (Left side pane): It has a button for each available function: ST, RC and SR buttons (for store, recall and reading status register, respectively) are disabled letting its possible use in the future opened. There are a couple of buttons, SPI and I2C, at the pane´s top allowing the user to change the protocol/memory focus.
2. Variables (Central pane): It is composed by two sections: setters and text info areas showing current values for all variables that may be involved in the functions.
3. PWM duty call (Left bottom pane): A slider with values ranging from 0 to 255, which sets the value for the PWM analog writing and, hence, the variable voltage value. It is scaled, so value 255 would correspond to 3,3V and value 0, to 0V. It must be taken into account that, depending on the technology, manufacturing process, particular chip, etc., the voltage supply below which the memory cells do not retain data may vary. Utilizing a VCC level below this so-called “threshold voltage” is not advised since it leads to information loss.
4. Status (Right pane): It contains two text areas. Upper text area is a log. It displays every interaction performed including variable updates, functions calls and the output coming from the board as results and possible errors. The lower area is just a record for called functions. It only displays the functions executed to make easier the user to follow the execution flow.

All interactive elements have their corresponding action listeners that perform the operation desired. They send the command as a string to the board and also update de logs content.

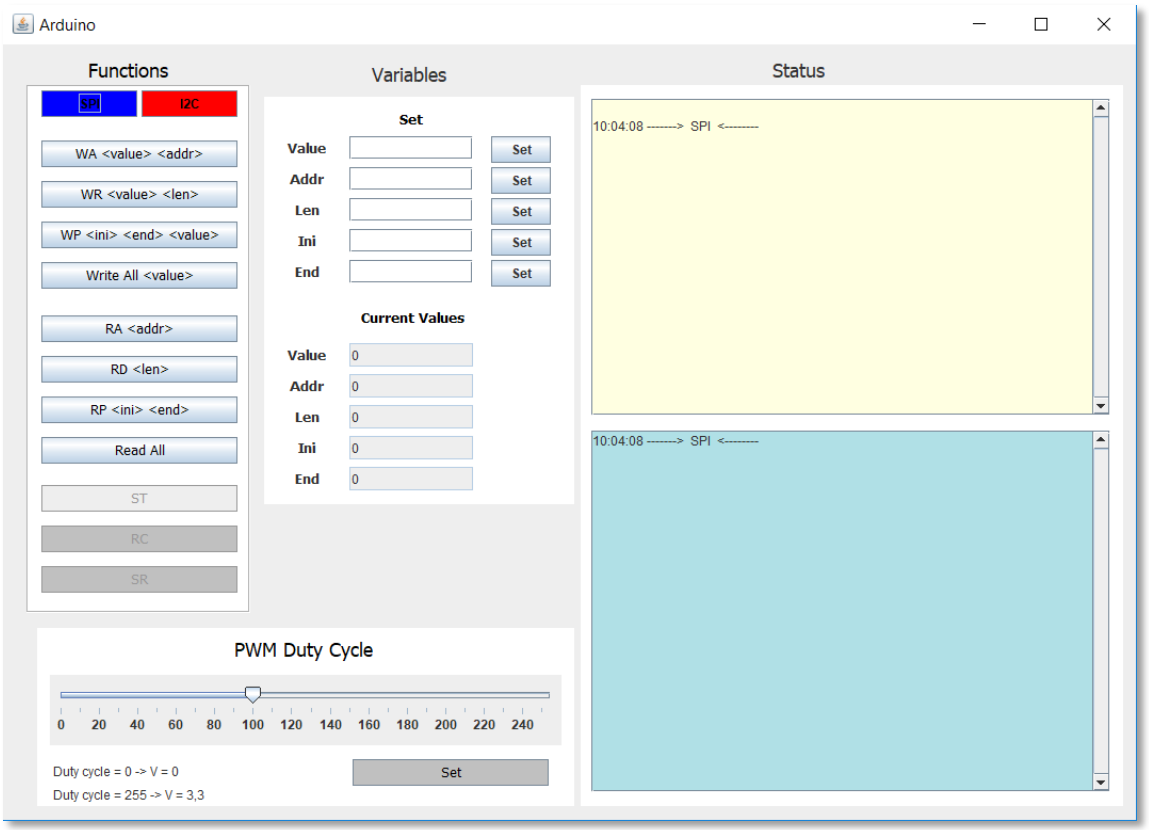

Figure 40: Java GUI.

Hardware design. Overall circuit description

As this work lays within a research project being carried out by the Computer Architecture Department at FDI, we had to meet certain constraints when designing the hardware. The first and more important one is the use of an Arduino Due, put in place by the department and used before in similar experiments.

The final requirements for the project were agreed upon after going through a series of meetings with professors Juan Antonio Clemente and Juan Carlos Fabero:

- The system will be composed of three elements: a computer, an Arduino Due and a memory board.
- The computer will run an interface to interact with the Arduino using USB.
- Use of CAT6 twisted pair to communicate with a daughter board, so the Arduino could reside in a different room.
- Variable voltage, rail-to-rail supply, for daughter board with a 40mA current rate. This voltage will be used to power up the memories.
- The daughter board should support three memories, Cypress CY14B101J – nvSRAM I^2C memory and Cypress CY15B102Q & CY15B104Q SPI memories.
- The footprint for the daughter board memories should be compliant with the SOIC to DIP adapters provided. SOIC8 to DIP8 for SPI and SOIC16 to DIP16 for I^2C.

- Daughter board memory placement should be centered over the Y axis and mounting holes for a servo should be provided.

Considering the low current delivered by the Arduino Due microcontroller, a bypassable line buffer was added to the SPI output. The block diagram of *Figure 41* shows the overall circuit elements designed to fulfill the project requirements.

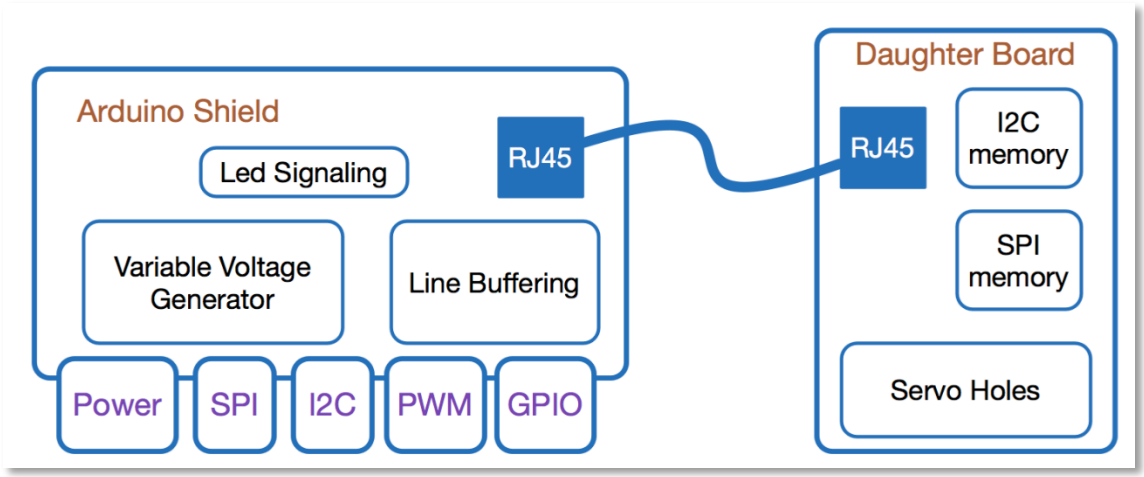

Figure 41: Circuit block diagram

This document describes in detail each of the two project boards, including design choices and calculations for components.

The complete rig for the memory testing cycle is presented in *Figure 42*.

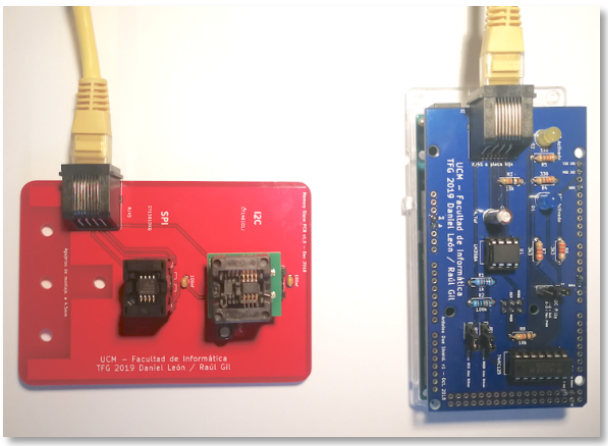

Figure 42: Complete testing custom-hardware. Daughter board (red) to the left and Main shield board (Blue), mounted over the Arduino Due, to the right.

Arduino shield PCB design

The main PCB of this project is the Arduino shield PCB. It supplies the variable voltage, routes the SPI and I^2C signals and buffers the SPI output signals, through its RJ45 connector and a CAT6 twisted pair cable, to the daughter board.

The name “Shield” is native to the Arduino ecosystem and refers to the layout that a PCB must meet to plug on top of the Arduino board. A full Arduino Due shield layout was put in place for the main board, mainly for completeness and future expansion.

The overall schematic for the main shield board is presented in *Figure 43*.

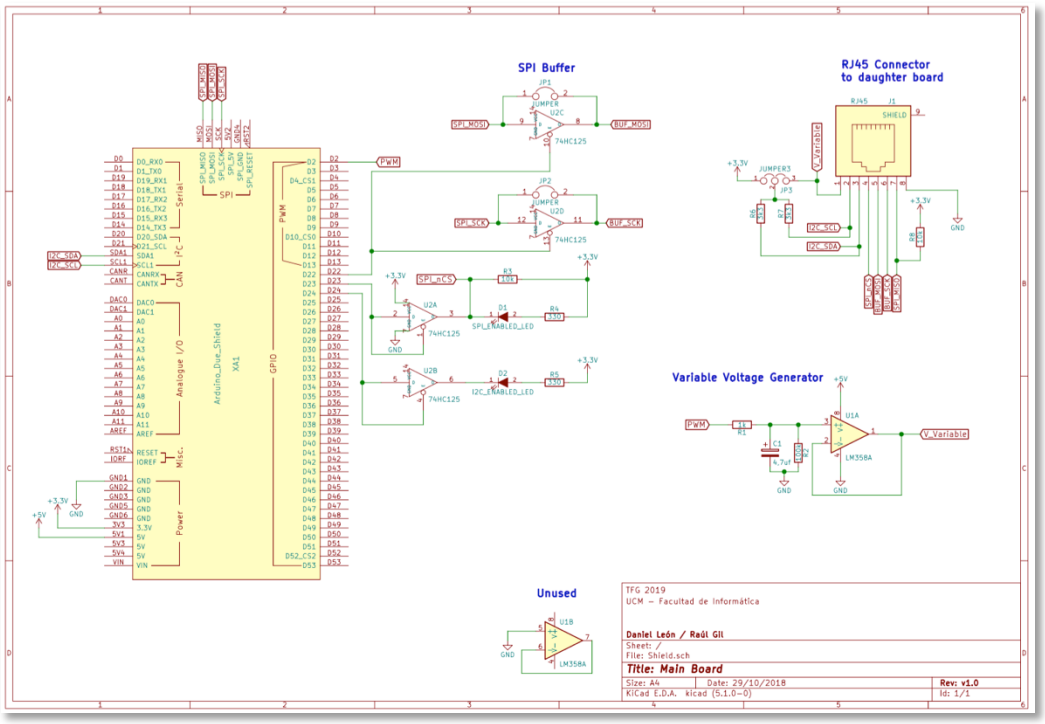

Figure 43: Main PCB - Arduino shield schematic

The LM358A includes two operational amplifiers, but the circuit only uses one. Configuring the unused one as shown at the bottom of the above diagram, ensures a noise-free operation from the active OpAmp.

Variable voltage generator

One of the requirements for the system is being able to feed the memories with a variable voltage. We were presented with two options evaluated in the past to fulfill this goal:

- Use internal DAC of Arduino Due for voltage generation.
- Use an external variable voltage source and attach it to the shield.

Both options were evaluated during the project launch meetings and both were rejected due to serious limitations.

Using one of the internal DACs existing in the Arduino Due for generating a voltage presents the limitation of the voltage range not being rail-to-rail, this is, Arduino Due’s DACs can only deliver variable voltage from 0.55V to VCC-0.55V, which was not in line with the project requirements. Additionally, although not fully specified in the Due’s documentation, DAC’s pin are low-current pins and can only source up to 3mA. Further

investigation showed that this is the safe amount of current a DAC can deliver, way below the required 30-40mA for the memories.

On the other hand, an external variable voltage source had two limitations that we did not want to impose to the project. The first one was the actual need for the external power source, whereas the other was the impossibility to include automatic voltage changes in the testing procedures.

Therefore, we had to devise a new solution to deliver the variable voltage and we decided to make use of the PWM capability of the Arduino Due. The PWM output was combined with a first-order low-pass filter for smoothing the signal and an operational amplifier in non-inverting, unity gain configuration. *Figure* 44 shows the circuit diagram.

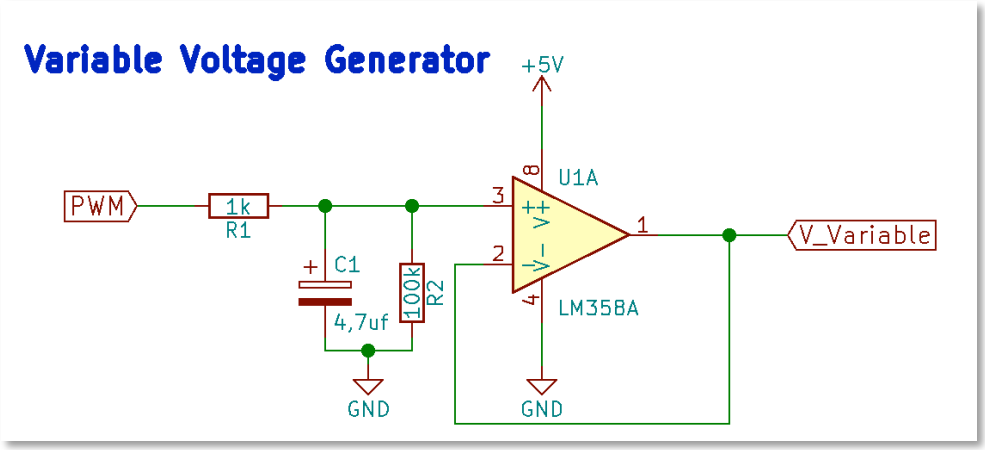

Figure 44: Variable voltage generator based on PWM, Low-pass filter for ripple rejection and OpAmp

The objective of the smoothing filter is to generate a DC voltage from an AC voltage. The PWM output generates an AC-like voltage with a square wave oscillating between 0V and 3.3V. This signal is then filtered by resistor R1 and capacitor C1. Resistor R2 is a bleeding resistor, in place to discharge C1 in absence of signal (PWM pin in HiZ) so it is ensured that the V_Variable output is 0V on boot.

To avoid the inclusion of a big capacitor in the circuit, which would cause slow response to the variable voltage selection, a moderately high PWM base frequency of 31.25KHz was initially selected.

The filter must reject most of the signal ripple in order to obtain an almost plain voltage dependent on the PWM duty cycle. The RC filter is then determined by first fixing the R1 resistor value. For these calculations, the R2 bleeding resistor is dismissed, given the order of magnitude of its value in relation to R1.

PWM is coming out from a high-current pin of the Arduino Due, so it can source up to 15mA and sink up to 9mA. Provided that the pin voltage is 3.3V, the minimum resistor to limit the current is 366Ω. To ensure an operation well within tolerances, a 1KΩ resistor, limiting the current to 3.3mA, was chosen.

Given the capacitor charging curve is characterized by $V_c = V_s \times (1 - e^{\frac{-t}{R \times C}})$ and its discharging one, by $V_c = V_{init} \times \left(e^{\frac{-t}{R \times C}}\right)$, and knowing the base PWM frequency[33], C can be calculated based on the fixed R and the desired ripple.

A target ripple of 5mV is set, considering this is well within the ripple acceptance of the memories used and less than the typical power brick ripple.

When selecting a smoothing capacitor, the RC constant should be much higher than the period of the signal and, since the target ripple is much smaller than the source voltage, a linear approximation could be used:

$$V_{pp\_ripple} = \frac{I_{load}}{2 \times f \times C} \Rightarrow 0.005 = \frac{1.65/1000}{2 \times 31250 \times C} \Rightarrow C = \frac{0.00165}{2 \times 31250 \times 0.005} = 5.28\mu F$$

Note that the calculation is considering a capacitor voltage of 1.65V at 50% duty cycle[33], therefore, the charge intensity is $\frac{V_{Hpwm} - V_{capacitor}}{R_{load}} = \frac{1.65}{1000} A$

Verifying the result using the discharge capacitor formula applied to a 50% charged capacitor by a PWM duty cycle of 50% to calculate C for the desired ripple:

$$V_{cap} - V_{ripple} = V_{cap} \times e^{-\frac{p_{pwm}/_2}{R \times C}} \Rightarrow \frac{V_{cap} - V_{ripple}}{V_{cap}} = e^{-\frac{p_{pwm}/_2}{R \times C}} \Rightarrow$$

$$\Rightarrow \frac{1.645}{1.65} = e^{-\frac{0.000016}{1000 \times C}} \Rightarrow \ln\left(\frac{1.645}{1.65}\right) = -\frac{0.000016}{1000 \times C} \Rightarrow C = \frac{0.000016}{1000 \times 0.003035} = 5.27\mu F$$

Where P_{pwm} is the period of the PWM signal, V_{cap} is the current capacitor charge and V_{ripple} is the desired ripple. It was proven then that the approximation was very good, as expected since the target ripple is almost a thousandth of the source voltage.

Given the closest commonly available commercial capacitor is 4.7µF, this is the value used in the circuit, obtaining a ripple of 5.5mV. (5.1µF capacitors are commercially available but, usually, at backorder).

As an example, *Figure 45* shows the ripple in the signal, at 50% duty cycle, for a capacitor of 0.33µF. For the oscilloscope capture image, the input signal is shown in yellow, at 1V division, while the output signal is shown in blue at a 20mV/div. Time division is 10 microseconds.

A 0.33µF capacitor shows a theoretical 79mV ripple (left) and a 74,4mV ripple in the oscilloscope capture (right).

[33] Voltage in the capacitor for a n% Duty Cycle PWM signal is V_{Hpwm}*n%.

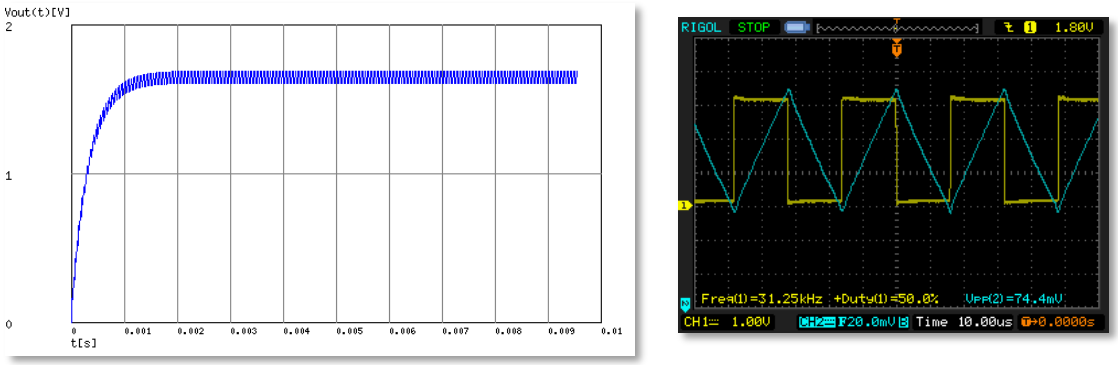

Figure 45: Variable voltage ripple for a 0.33µF capacitor. Simulated vs. measured

The chosen capacitor is a 4.7µF. With a theoretical ripple of 5,5mV, it shows the measurement results depicted in *Figure 46*.

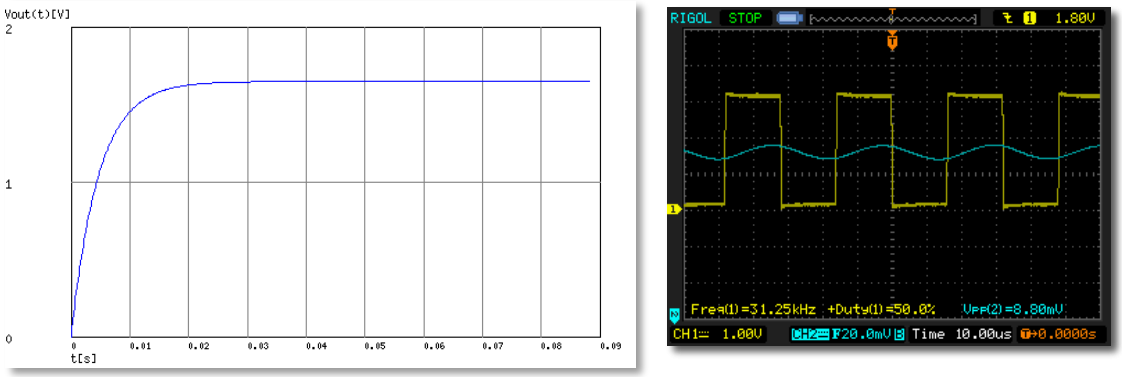

Figure 46: Variable voltage ripple for a 4.7µF capacitor. Simulated vs. measured

Note that the actual measured ripple being 8.8mV is within the expected variance for components values (R and C), the inclusion of the bleeding resistor and the sensitivity of the oscilloscope ADC (Analog to Digital Converter).

Therefore, the selected **RC filter is R=1000Ω and C=4.7µF.**

Once the RC values are fixed, the time to reach 3.3V can be calculated. This is important since the Arduino software must ensure a delay of this value, after enabling the PWM at 100% Duty Cycle, before attempting to communicate with the memories.

When the source voltage is stable, the time constant τ of the circuit is defined by RC. τ = R x C = 1000 x 0.0000047 = 0.0047 seconds. A capacitor reaches its full charge at 5τ = 5 x 0.0047s = **23.5ms.**

Line buffering and signaling

A line buffer for the output SPI signals (SCK, MOSI, SS) has been added to the main board foreseeing a higher current usage in future expansions of the project. Arduino Due's SPI ports only deliver up to 3mA of current, seriously limiting the potential fan-out of the signals. Therefore, tri-state buffers have been included in the design. The chosen part is the 74HC125[34] Integrated Circuit, featuring four tri-state, non-inverting line buffers with a typical 10ns transition time at 3.3V and a maximum delivered current of 80mA.

The transition time limits the effective SPI frequency to 50MHz, whereas bypassing the buffers will provide an SPI frequency only limited by the master and slave devices and the cable resistance-capacitance factor. Even though this is higher than the current selected memories maximum SPI speed, a bypass has been put in place on the main PCB for the MOSI and SCK signals (JP1 and JP2, respectively, see *Figure 47*), in case a future part requires higher frequencies.

The high output current capability of the buffer has also been harnessed to enable LED signaling for the bus activity, connecting a LED to the SS/CS (Slave Select) buffered signal and using the fourth, formerly unused, buffer to connect an I²C activity indicator LED.

The I²C bus does not require any buffering as it is open-drain/open-collector. Arduino Due's I²C ports are low-current, sinking a maximum of 6mA. A standard 3.3KΩ resistor value is used to pull-up both SDA and SCL, for a maximum sunk current of 1mA.

Figure 47, to the right, shows the detail of the line buffering and signaling.

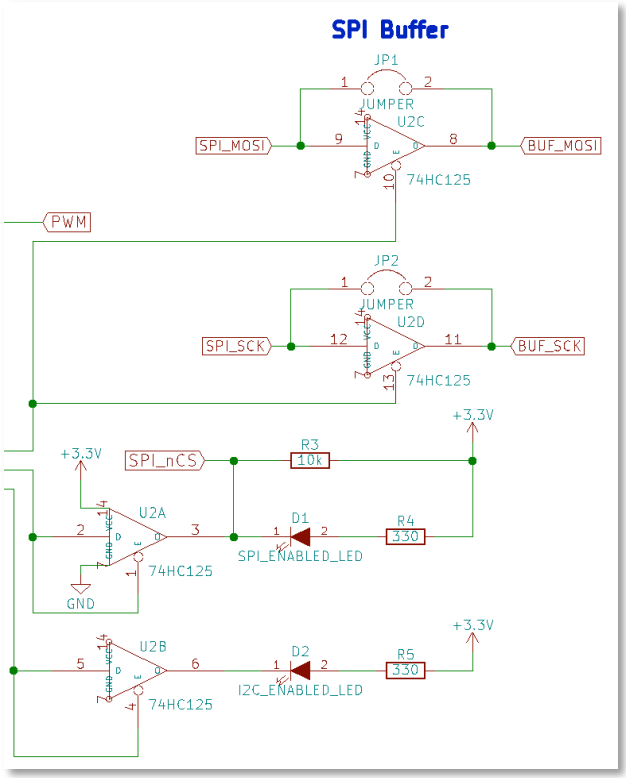

Figure 47: SPI Buffering and LEDS of the main PCB

[34] 74HC125 Datasheet: https://assets.nexperia.com/documents/data-sheet/74HC_HCT125.pdf

I²C pull-up selection – Operation at lower voltages

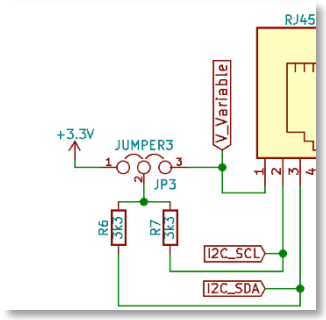

Figure 48: I²C pull-up selection

In this design, it has also been ensured that the I²C SDA and SCL signals can be pulled-up to either 3.3V or to the variable voltage rail, enabling I²C memory operation at lower voltages.

Said selection is done by configuring the jumper JP3, enabling 3.3V or the variable voltage as the VCC rail for the I²C bus pull-up as shown in *Figure 48.*

Note that SPI operation at lower voltages is not permitted under normal conditions, as the SPI signals remain at nominal VCC levels. Operating SPI with a lower variable voltage feeding the memory may damage and/or destroy the memory as stated in the maximum ratings of the parts datasheet. For normal operation, tri-state buffers should be disabled when lowering the voltage on a SPI memory. However, since the goal of the system is test memory chips in different conditions, this is neither limited by the hardware nor the software, and using the correct configuration is left to the user's discretion.

Memory Daughter board

A daughter board is designed to place the memories in front of the radiation beam. It attaches to the main board by a CAT6 twisted pair cable. The goal of this board is to physically align the memories on the X axis, so they can rotate to receive neutron impacts at different incident angles. Being a purely passive board, it just routes the signals to the appropriate pins of the memory sockets.

Our board is designed using ZIF (Zero Insertion Force) memory sockets, so it would be fast and easy to exchange memory parts once irradiated. In addition, it features industry-standard distanced mounting holes for attaching the PCB to a servo motor, so rotation might be automated during the radiation process.

Figure 49 is the schematic for the daughter board. Note that C2 and C3 capacitors are decoupling capacitors to filter any potential high-frequency noise that may arise during the radiation process. The 47µF capacitor labeled C1 is in place to enable the "power failure Quantum Trap" as specified in the CY14B101J datasheet, allowing for transferring the contents of the memory to the non-volatile elements in case of an unexpected power shutdown. All pins tied to ground or left unconnected correspond to the specifications for each part as printed in the latest datasheets.

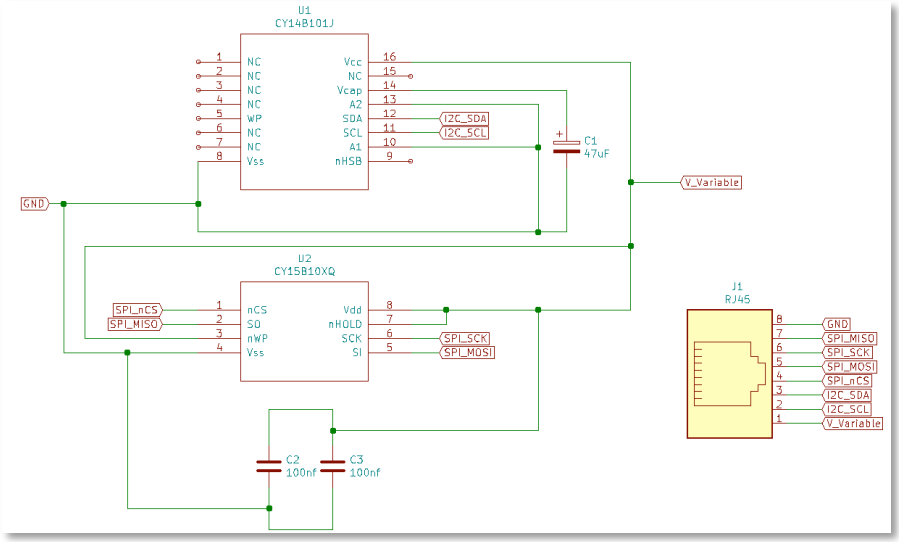

Figure 49: Daughter board schematic

Twisted pair and RJ45 connector

Both boards are designed to be interconnected by a direct (non-crossover), CAT6 twisted-pair cable with RJ45 male plugs, following the ANSI/TIA/EIA-568 -B[35] wiring standard as shown in *Figure 50, to the right*. Each board has a female RJ45 header routing the signals from/to the correct pins. By using this standard, interconnection of both boards may be done by using any common Ethernet cable.

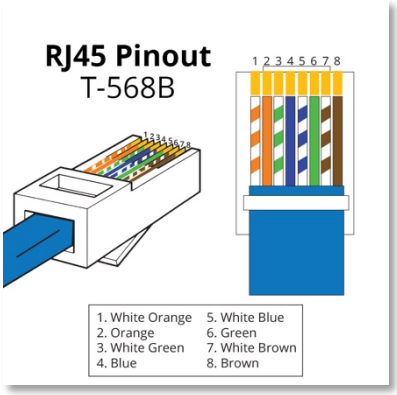

Figure 50: T-568B wiring Standard - Integrated Network Cables Inc.

[35] Telecommunications Industry Association: https://www.tiaonline.org

BIBLIOGRAPHY AND HYPERLINKS

Component Datasheets

CY14B101J Datasheet: https://www.cypress.com/file/44701/download
CY15B102Q Datasheet: https://www.cypress.com/file/175731/download
CY15B104Q Datasheet: https://www.cypress.com/file/209146/download
MC68HC11A8 Datasheet: http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC11A8.pdf
LMH1218 Datasheet: https://www.ti.com/lit/ds/symlink/lmh1218.pdf
74HC125 Datasheet: https://assets.nexperia.com/documents/data-sheet/74HC_HCT125.pdf

Protocol specifications, manuals and documentation

I^2C bus user guide: https://www.nxp.com/docs/en/user-guide/UM10204.pdf
I^2C Manual (AN10216-01): https://www.nxp.com/docs/en/application-note/AN10216.pdf
SPI reference (Chapter 6): http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC11A8.pdf
Arduino Reference for SPI: https://www.arduino.cc/en/Reference/SPI
UART paper: https://www.researchgate.net/publication/308988751_Universal_Asynchronous_Receiver_and_Transmitter_UART

Arduino related

Arduino website: http://www.arduino.cc
Arduino Due: https://store.arduino.cc/due
Arduino IDE: https://www.arduino.cc/en/Main/Software
Wire Library: https://www.arduino.cc/en/Reference/Wire
WireBeginTransmission() Function: https://www.arduino.cc/en/Reference/WireBeginTransmission
WireEndTransmission() Function: https://www.arduino.cc/en/Reference/WireEndTransmission
WireWrite() Function: https://www.arduino.cc/en/Reference/WireWrite
Arduino Reference for SPI: https://www.arduino.cc/en/Reference/SPI
Panamahitek library: http://panamahitek.com/libreria-panamahitek_arduino/
Cyclic Executive definition: https://en.wikipedia.org/wiki/Cyclic_executive

Tools

KiCad: http://kicad-pcb.org
Gerber official website including specification: https://www.ucamco.com/en/gerber
Eclipse IDE: https://www.eclipse.org/ide/
Eclipse user license: https://www.eclipse.org/legal/epl-2.0/

Books

ISBN-13: 978-0132622264 / ISBN-10: 0132622262
Electronic Devices and Circuit Theory (11th Edition) – Robert L. Boylestad, Louis Nashelsky

ISBN 84-8041-022-1
Manual de matemáticas para ingenieros y estudiantes – I. Bronshtein, K. Semendiaev

Other references

Scrum methodology: http://www.scrum.org
Telecommunications Industry Association: https://www.tiaonline.org
SeedStudio PCB manufacturer: http://www.seeedstudio.com
JLCPCB PCB manufacturer: http://www.jlcpcb.com
DirtyPCBs PCB manufacturer: http://www.dirtypcbs.com
Telegraph Patent (Async. Serial):
https://worldwide.espacenet.com/publicationDetails/originalDocument?CC=US&NR=1199011A&KC=A

INDEX OF FIGURES